IEEE

MICRO

JUNE 1989

Chips, Systems, Software, and Applications

SPECIAL FAR EAST ISSUE

The BTRON / 286 applies the TRON Project

HUMAN-MACHINE INTERFACE

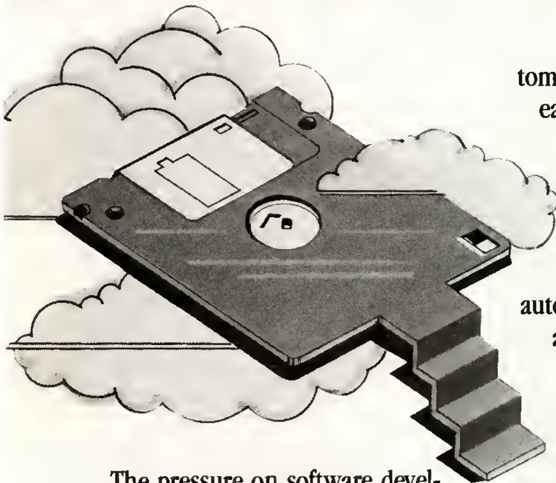
PLUS
- An FPU for TRON
- A Data-driven Processor for Consumer Electronics

SPECIAL FEATURES
- A Logical Tool for Relational Databases
- The Transputer T414 Instruction Set

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Your Stairway To Software Heaven.



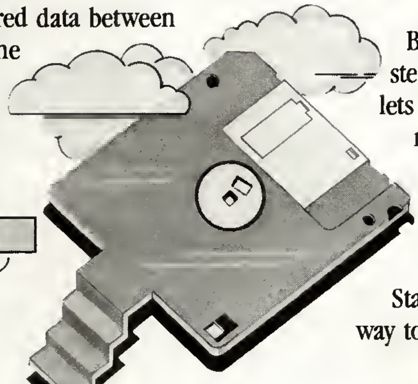
The pressure on software developers to *produce* has never been greater. Yet there they sit, often reinventing the wheel, with more computational power than ever at their fingertips and the clock ticking away. Product delivery deadlines? So much pie in the sky.

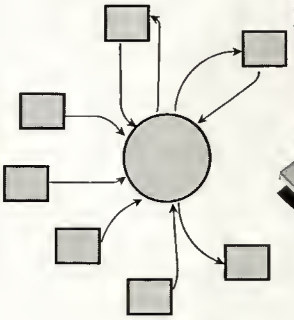
The problem? How *best* to put all that PC CPU capacity to good use.

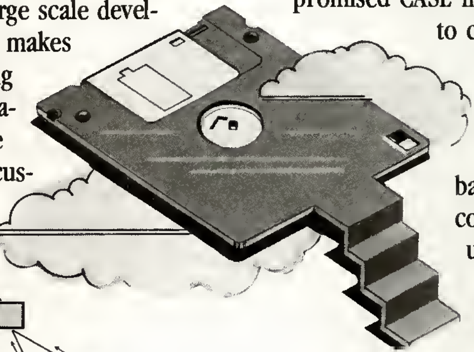
The solution: the **Visible Analyst Workbench.**

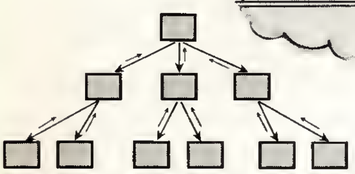
The Visible Workbench makes the full power of CASE accessible to *everyone*. Running as a multi-user tool on Novell LANs or on individual PC workstations, the Visible Analyst Workbench lets teams of software engineers work together — and *simultaneously* — on large scale development projects. It makes after-the-fact piecing together of specifications a thing of the past. And, as our customers delight in telling us, it's so easy to learn and use that people begin working more productively on "day one."

The Visible Analyst Workbench delivers *real* development power. The power of automated, linked structured analysis and design. The power of an automated data repository. The power of prototyping. The power of instantaneous communication and shared data between project members. The power of accurate, validated high level specifications with full documentation. And, soon, bridges to code generation, completing the promised CASE link "from pictures to code".

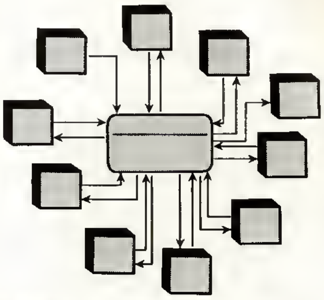








In fact, the Visible Analyst Workbench is the *only* PC-based CASE tool that combines ease of use, self-implementation, cost effectiveness and *true* multi-user capabilities. And, because it *is* a CASE tool, its usefulness will seem everlasting.



Best of all, our step-by-step product growth path lets you begin building internal CASE resources at down-to-earth prices starting under $300.

The Visible Analyst Workbench.

Start building *your* stairway to software heaven today.

Down to earth prices

Professional Series — For large, multi-project systems development:	
3-Node LAN Pak	$ 3,500
per additional node ..	700
Stand-alone version ..	1,785
with Prototyper	2,380
Personal and Educational Series — For small project development and educational needs.	
Personal Edition	695
Educational and Training Version	295

Visible Systems
CORPORATION

The Bay Colony Corporate Center • 950 Winter St. • Waltham, MA 02154 USA
(617) 890-CASE FAX (617) 890-8909 Telex 261102 VSCUR

Reader Service Number 2

* Submit six copies of all articles and special-issue proposals to Joe Hootman, EE Dept., University of North Dakota, PO Box 7165, Grand Forks, ND 58202; (701) 777-4331 Compmail+ j.hootman

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, California 90720
(714) 821-8380

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578; IEEE Computer Society Headquarters, 1730 Massachusetts Ave., NW, Washington, DC 20036-1903; IEEE Headquarters, 345 East 47th St., New York, NY 10017. Annual subscription: $18 in addition to IEEE Computer Society or any other IEEE society member dues; $33 for members of other technical organizations. Back issues: $10 for members; $20 for nonmembers. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to *IEEE Micro,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second-class postage is paid at New York, NY, and at additional mailing offices.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society. Send editorial correspondence to *IEEE Micro,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations. *IEEE Micro* subscribes to the Computer Press Association's code of professional ethics.

Cover illustration by Hajime Sorayama and Image Bank West
Cover design by Hayden Design and Design & Direction

IEEE MICRO

Volume 9 Number 3 (ISSN 0272-1732) June 1989

Published by the IEEE Computer Society

Departments

Feature Articles

Special Features

Letters

RISC tutorial comments

To the Editor:

I'm sorry, but I'm about to become one of those guys who forgets to tell you when you've done a great job and only complains when something bugs him enough.

What bugged me enough was Lazzerini's RISC article in *IEEE Micro*, Feb. 1989, pp. 57-65. I can't quite see why you found this article worthy of publication. There isn't one new idea in it. It's a rehash of things that were all said before by more eloquent voices. It seems to me that anyone who wants to discuss RISC nowadays must concentrate on machines such as the MIPS R3000, HP Precision, [Sun Microsystems] Sparc, AMD Am29000, Motorola 88000, etc. If one wants to go over thoroughly trod ground such as that represented by the early academic RISC machines, it would behoove one to bring some new and original insights to the treading. If there was anything original in Lazzerini's article, it escaped me.

Lazzerini didn't even define her terms, which could mislead the reader into thinking there is actually some consensus on what RISC means. Personally, I think there is an emerging consensus, but it appeared nowhere in Lazzerini's article, nor [did] any mention of why it is that machines that all claim the RISC label can differ as much as the AMD, Motorola, MIPS, and Sparc do. This article was just not good enough and shouldn't have appeared in *Micro*. But thanks for the others, they were worth reading.

Robert P. Colwell
Branford, CT

To the Reader:

IEEE Micro also received a letter from Daniel Tabak concerning the same article. He has kindly allowed us to edit the letter and approved the editing. Tabak makes the following points:

The material in the Lazzerini article is available from other sources such as a tutorial by Stallings, "Reduced Instruction Set Computer Architecture," *Proc. IEEE*, Vol. 76, No. 1, Jan. 1988, pp. 38-48; and a book by Hayes, 2nd ed., McGraw-Hill, New York, 1988.

There are several sources of RISC processors available on the European market, which are good RISC models (Transputer, Acorn, Metaforth) and which were not covered in this article. More information can be found on these processors in Tabak's *RISC Architecture*, Research Studies Press Ltd., UK, and John Wiley & Sons, New York, 1987, and in an article entitled "RISC Systems" in *Microprocessors and Microsystems*, Butterworth & Co., May 1988, by Tabak.

The Katevenis dissertation was published as a book by MIT Press, Cambridge, Mass., in 1985.

Lazzerini's "The RISC Approach" drew some other responses. Of the 26 other readers who have read and assessed the Special Feature article so far, three rated it as being of low interest, seven rated it of moderate interest, and 16 rated it of high interest. Some other readers commented on the article; their comments appear in this issue under the In the Mailbag heading on p. 6.

We welcome other comments from readers on this and any article appearing in *IEEE Micro*.

Joe Hootman
Editor-in-Chief

From the Editor-in-Chief

I get letters...

This year I have been impressed with how much mail has crossed my desk. I am sure that in previous years I have had just as much, but for some reason the mail seems to be out of hand this year. I very informally surveyed the mail that I received by saving two weeks of mail that came to me during the first part of March. In this two-week period I found that I'd received 88 ads in the form of letters, flyers, and brochures; 23 newspaper publications; six plastic-bagged cards; and 36 magazines, five of which were IEEE publications. I did not count local and job-related memos or the mail that I receive at home.

If my mail is typical, a substantial amount of material is being produced for people to read and absorb. The real question is how publications like *IEEE Micro* stay in business and hold their market share.

Well, *Micro* has several things going for it that make it a quality publication, and I think—everything else being equal—people will naturally migrate to a quality product. Every article that appears in the publication is reviewed by volunteers in the Computer Society. I send new manuscripts to one of the *Micro* editorial board members, who then selects three peer reviewers. These reviewers formally assess each manuscript for its technical content, originality of ideas, clarity of thought, and reader interest. They may suggest additions, deletions, major rewriting, or, as happens in a surprising number of cases, outright rejection. Clearly, the editorial board upholds *Micro*'s high standards and exerts much influence over the technical content of each issue.

The results of each review are given to me, and I in turn inform the author of the results of the review. After an author has satisfied the reviewers' recommendations, the manuscript is ready for publication and earmarked for either a particular theme issue or the first-available opening as a special feature. Every accepted paper next passes through the editing process in the Computer Society's Los Alamitos office. Either Marie English or Christine Miller works with the authors to produce a clear, articulate article that can be understood by both experts and novices in the industry. Readability is a top concern throughout this process. The quality of the editing is apparent in the publication, but it is really appreciated by the authors as well. Marie and Christine have received excellent reviews from our authors.

The whole manuscript process benefits readers. Personally, I tend to read material that is current and that has had some editing and reviewing. I find that the editing process tends to filter out the unnecessary and eliminate some of the trivial. I suspect that most readers feel the same.

Micro's departments have also been well received by a great number of readers. These departments play an important role in dispensing industry knowledge and are a tribute to the board members who generate them. Readers have indicated the popularity of such departments as New Products, Micro Review, Micro World, Micro Law, Micro View, and Micro News. All of these departments are edited.

Lastly, *Micro* brings to its readers international news from Europe and the Far East in the form of two special issues each year from these regions.

The current issue is edited by Ken Sakamura (University of Tokyo) and is one of the special issues reflecting ongoing work in the Far East. Ken is the father of the TRON concept as well as being an *IEEE Micro* editorial board member. *Micro*, as you may recall, was the leader in the introduction of the TRON concept to the United States reader. The project is interesting in that it has the potential to serve an international as well as a regional market. It is increasingly clear that in order to serve a large regional market efficiently, it is necessary to also serve a wider international market. A company can attempt to do this in two ways, by default or by design. TRON is an attempt to serve a world market by design. If TRON does succeed in the international market, the impact on the computer market in the United States will be substantial. It will be worth watching the development of the world market to see how TRON and TRON-related products fare.

In addition to the Special Far East ar-

ticles selected by Ken Sakamura, you will find an article by Jean-Daniel Nicoud and Andrew Tyrrell on the Transputer 414 instruction set. The transputer is a unique device that has changed, to some degree, the way we view parallel-computing hardware. Here's a chance to look at the instruction set of this most interesting device. A second article written by Mehdi Owrang and Gemini Gunaratna discusses a logical design tool for analyzing data on a logical as well as a conceptual level. The concepts presented in this article allow the design of large relational database systems.

J. Hootman

In the mailbag

October 1988

"I would like to see articles on advanced inventory control, personnel, and payrolls when some 400 variables are to be managed." M.K.D.A., Mosul, Iraq (Several journals and magazines evaluate and discuss the various aspects of business software, for example, *Lotus*, *PC Magazine*, and *PC World*. *IEEE Micro* does not see these topics as areas that are of interest to most of its readers.—J.H.)

December 1988

"I liked everything; it seems I found the right engine to power our graphics system. DSPs are my discovery of the month. I would like to see more on modern CPUs including RISCs and evaluation of performance of CPUs and systems. . . ." G.M., Warsaw, Poland (I am pleased that you found the special issue on signal processing of interest. The DSP issue is one of the most popular issues of *Micro*.—J.H.)

"I liked the article on the TMS320C30 floating-point digital signal processor." J.O.S., Makati, Philippines

"I liked the articles on DSP." A.P.T.G, Christchurch, New Zealand

"I liked the TMS320C30 floating-point DSP article. I liked the article on a PC-based digital speech spectrograph." A.N., Isfahan, Iran

"I liked the DSP reviews. The benchmarks for all the DSPs ought to have been the same, which would have made comparison easier. The benchmark table for the DSP96002 is very good. I would like to see prices and ordering information for the reviewed products (DSPs for example). This is very useful for foreign countries (like India)." S.N., Pune, India. (Until manufacturers agree on common definitions that characterize their hardware and software and these definitions are universally accepted, prediction of device performance when compared with other devices is at best a risky business. As things stand now, it is not possible for us to edit tables of data to a standard.—J.H.)

"I liked the whole issue, especially the articles on the TMS320C30 and the DSP96002. I would like to see more about Motorola's new DSP chips." C.R.G., Acassuso, Argentina

"I liked the whole issue. I would like to see even more on DSP hardware and DSP algorithms." S.G., Newcastle, England

"I would like to receive more information on the subject on page 68 ('A PC-based Digital Speech Spectrograph') that includes the Radix FFT in the TMS32010 Assembly Language." F.H.T., Baghdad, Iraq (I have sent your request to the author.—J.H.)

"I would like to see single-chip microcomputer applications, low-power applications, the Motorola 68HC11 for example." A.K., Calgary, Alberta

"I would like to see more on CNC, process control/simulation, and factory automation." A.B., Yverdon, Switzerland

"I thoroughly enjoyed the DSP issue for updating on current processor status and architecture. There are always very diverse topics presented (in *Micro*), which is good for broadening one's background. Keep up the good work." G.L.S.C., Portland, ME

"I liked the entire issue and I would like to see more." K.S., Nepean, Ontario

February 1989

"I liked the magazine. I only wish I had time to read it from cover to cover." T.B., Victoria, British Columbia (I am pleased that you take the time to read at least part of *Micro*.—J.H.)

"I liked the real-time implementation of the Newton-Euler equations of motion on the NEC µPD77230 DSP." M.J.T., Highland, MI

"Thanks for a job well done, Jim; Joe, welcome! I would like to see authors' full addresses with articles. I would like to correspond sometimes. . . ." A.D.W., Lewisburg, PA

"Nice wording on the lead-in to Micro View interview! Top quality; nice job, Marie and Christine." J.S., Phoenix, AZ (High praise indeed when coming from our magazine's first managing editor. Thank you.—J.H.)

"I disliked 'The RISC Approach.' It did not include the 286/386 while including the Z8000, which makes the article suspect." D.F., Fort Wayne, IN (See Letters to the Editor in this issue.—J.H.)

"I liked the RISC approach; good overview of approach. The inability of CISCs to expose microcode sequences for compiler optimizations is an often-overlooked point." J.B.H., San Jose, CA (See Letters to the Editor.—J.H.)

Micro World

Hubert Kirrmann
Asea Brown Boveri Research Center
CRBC.1
CH-5405 Baden, Switzerland

Neural computing: The new gold rush in informatics

In some abandoned mine, one stubborn miner hits pay dirt, and a new gold rush starts. This miner will become rich, but many others will never do so. What happened recently to the abandoned technology of LOX (liquid oxygen temperature) superconductivity currently repeats itself with neural computing. The neural computing technique, buried 20 years ago after seemingly unimpeachable proof of its inability, emerges now as a promising new direction in information technology, or informatics.

Basically, neural networks mimic the structure of the neural system of animals. Elements with multiple inputs, which change state (fire) when the sum of their inputs exceeds a certain threshold, emulate the system's neurons. They can be implemented by an analog summing amplifier with a threshold function, the inputs being connected over resistors.

Neurons are organized in connected layers; the output of each neuron of a layer connects to the input of each neuron of the next layer. (Some networks also connect neurons within the same layer; others consider symmetrical connections.) The number of connections rises of course with the square of the number of neurons per layer.

The strength of each connection is assigned a weight. These weights may be modified to give the network certain capabilities, for example, to recognize patterns applied at the input layer. Contrarily to classical computers, which are programmed, neural networks must be taught; they require a learning phase. Numerous different algorithms of learning exist, which dictate how the weights are modified in response to examples presented to the input layer of the network. Though these structures have been known for a long time, the breakthrough came from new learning algorithms that allowed us to teach intermediate (hidden) layers of neurons.

The first conferences on neural networks attracted an incredible number of believers in the new branch of progress. Proceedings of more than 2,500 pages pile up. In Europe two conferences, Neural Networks from Models to Applications (IDSET, Paris, June 1988) and Connectionism in Perspective (Zurich, October 1988), were very well attended, especially with respect to conferences on the classical computing fields.

Informaticians and engineers are only a minority of the participants. Others represent domains as far apart as zoology, drug-addiction psychology, brain research, thermodynamics, theoretical physics, mathematical statistics, and artificial intelligence. Expectations are high, confusion reigns, and concrete results are few.

Basically, we are still at tutorial time. Tutorials on neural networks used to begin with a description of the biological nervous system—which makes any neurobiologist unhappy. It is not just a matter of professional jealousy, but the fact that neuro-informaticians tend to fall in love with their biological model, although the methods they use often have no counterpart in nature. This fact is especially true for the very hypothesis that allowed the breakthrough in neural computing, namely that neuron connections are bidirectional and symmetrical. Nevertheless, it does not prevent neuro-informaticians from using biological terms with profusion, like "rewarding the network" or "killing neurons." Personification of computing systems often becomes a sign of exaggerated expectations.

Just as with rule-based artificial intelligence, the technological wave will settle down, and neural networks will complement classical computing and rule-based experts in some domains, rather than replace them. The domains of choice for neural networks are those in which rules are difficult to establish, either because they are unknown or too complex. The task of the neural network is to find rules and discover patterns in seemingly unorganized data. Pattern recognition for voice, image, or signals is already a mainstream technology.

Neural networks will also help find rules in complex data sets, which will be used as a base for other algorithms. A special domain is adaptive process control, which is in fact the only branch of informatics that never gave up the principle of analog computation. The backtracking algorithm, for instance, originated in adaptive control.

Current examples of neural networks are still small in scale. They face the same questions expert systems once faced: How do they scale when the problem becomes complex enough to be interesting? Can one really build a network of interesting size? How long does it take to program a network? Can learning time be reduced by mapping weights from another network, or must each network be taught independently?

Let's divide the domain of research into an algorithmic part, one that represents knowledge, another that simulates networks, and a final one that realizes networks.

0272-1732/89/0600-0007$01.00 © 1989 IEEE

• In the algorithmic part, we get most ideas from theoreticians and mathematicians. Theoreticians closer to the world of thermodynamics than to informatics rediscovered neural networks.

• The representation of knowledge, that is the mapping of the problem and its solution to the network, is closely related to the application. Here, many disciplines feel at home; everybody participates.

• Simulation of networks on classical computers is rather trivial, with most of the work occurring on the user interface. Numerous universities and research departments work on teaching aids. Simulators with a small number of neurons work fast enough to be used in real time (which makes them already a realization). The fact that sequential machines can simulate neurons makes it clear that parallelism and algorithms are separate issues. Some simulators are made on parallel processors, for instance on transputers, reflecting thus the parallelism of neural networks. The performance gain is not evident if one considers that most of the computing power is spent in communication.

• The realization of large neural networks in silicon brings tremendous challenges to hardware designers. While the neuron itself is rather simple, wiring is the problem. Nature knows ways to bus 10,000 signals to a single neuron, but we are unable to connect more than a hundred connections to a bus. The same problems appear today in classical radar signal processing, which requires the interconnection of some 20,000 processors. An alternative proposal could be optical neural networks.

In Europe, activities are numerous and some interesting results have already been achieved. The best-known work is possibly T. Kohonen's speech-recognition network based on the mapping principle developed at the Helsinki Institute of Technology. The European Community sponsors the BRAIN (Basic Research in Adaptive Intelligence and Neural Computing) and ESPRIT projects. The governments finance national projects, and the universities open faculty positions for neurocomputing.

Since it is still too early to describe works that most of the time are just in their infancy, I list only the research activities that may be of interest to the readers of *IEEE Micro*. I give only one contact person, and I'm sure I have omitted many works; please protest. Contacts between these institutes and readers of this column will help build a meganeuron network by increasing the reciprocal weights. Since there are too many to list, I depend on readers to locate addresses for contacts of interest.

I would like to thank J. Bernasconi of the Asea Brown Boveri Research Center who helped me explore the field of neural computing and prepare this discussion.

Neural Network Research Activities

Institute/Contact	Research specialty
Denmark	
Nordita, Copenhagen J.A. Hertz	Statistical learning Image processing
England	
University of Edinburgh D. Wallace	Statistical physical methods Associative memories Parallel processors
Imperial College London I. Alexander	Boolean neural networks
Royal Signals and Radar Establishment Malvern D.G. Bounds	Diagnosis Optimization Multilayer topological maps
Finland	
Helsinki University of Technology T. Kohonen	Topology-conserving maps Content-addressable memories Speech recognition
France	
E.S.P.C.I. Paris G. Toulouse	Associative memories Pattern recognition Spin-glass concepts Optimization
Ecole Nationale Superieure Paris J.P Nadal	Associative memories Learning
CEN-Saclay Gyf-sur Yvette B. Derrida	Associative memories
University of Paris V F. Fogelman	Feedforward networks Backpropagation learning Medical diagnosis
Ecole Polytechnique Grenoble C. Jutten	Signal procesing Associative memories
Germany	
Technical University Munich H. Ritter	Topology-conserving maps Motor task learning
University of Giessen M. Opper	Dynamics of learning Pattern recognition
University of Heidelberg J.L. von Hemmen	Associative memories Pattern recognition

Continued on p. 94

Micro Review

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705-1310
(415) 540-7745

The USA in today's world

At the end of World War II the USA's newly awakened economy stood in sharp contrast to the war-ravaged economies of the rest of the industrialized world. Military and industrial supremacy came easily. Perhaps complacency and laxity crept in. Self-examination and long-range economic, social, and environmental planning were not central to the country's life. Now, more than 40 years later the chickens have come home to roost.

Two recent books address the present American situation in positive ways. One of these, by Robert McIvor of Motorola, focuses specifically on what the semiconductor industry needs to do to compete in the world economy. The other, by Frances Moore Lappe of the Institute for Food and Development Policy, sees economic well-being arising from an active and open discussion of America's underlying assumptions and values. Interestingly, these two works, coming from strikingly different perspectives, lead us to confront the same basic issues.

Managing for Profit in the Semiconductor Industry, Robert McIvor (Prentice Hall, Englewood Cliffs, N.J., 1989, 544 pp.; $30)

If you believe, as I do, that Motorola has become a company capable of competing in the world semiconductor marketplace, you should be interested in Robert McIvor's account of how Motorola got that way and what it's doing to stay competitive.

McIvor ridicules the notion that we are simply moving into a post-industrial service economy in which the best jobs stay here, while low-level manufacturing functions migrate across the Pacific Ocean. He laughs at the idea that there is refuge in "niche markets" like ASICs. His view is that in a highly capital-intensive industry like semiconductor manufacturing, market share is a prerequisite to profitability. While everyone knows that market share depends on product utility, on-time delivery, price, and service, the successful Asian companies seem to have learned how to achieve these goals. American companies have some catching up to do.

According to McIvor the key to success is "world-class factories," by which he means an integrated process that encompasses manufacturing, marketing, and engineering. His book deals with that process:

> Its theme is transition . . . from economies of scale to economies of scope, from standardization and cost reduction to mix management and revenue maximization, from labor-intensive production systems to knowledge-intensive ones, from inventory-profligate functional manufacturing systems to just-in-time and short-cycle manufacturing methods, from inspecting-out-rejects quality systems to building in quality using real-time-feedback process control, from offshore escape to onshore renewal using the most contemporary manufacturing techniques. . . .

This covers pretty well the management mechanics aspects of the book, which comprise the bulk of the pages, but McIvor still has important things to say about people and attitudes. He sees the importance of motivation—not just the often negative motivation of "management" in the sense of control, coercion, pushing from behind, prescribing in detail, and so on, but the positive motivation of leadership. McIvor sees leadership as appealing to positive qualities: the need to accomplish and be appreciated, the need to belong to a group and contribute to a common enterprise.

Other people have said many of the same things. Often, however, the unspoken message that comes from watching to see who succeeds and who fails in an organization is different from the spoken message. Putting into practice the principles of motivation and leadership that McIvor advocates means creating a corporate culture within which everyone feels them as realities. There's no formula for how to do this, but I think that open and continuing discussion of fundamental beliefs and values is a prerequisite. That's why the next book is so interesting.

Rediscovering America's Values, Frances Moore Lappe (Ballantine, New York, 1989, 349 pp.; $22.50)

The Lincoln-Douglas senatorial debates of 1858 were wide-ranging affairs that went on for hours, but people followed them intently. According to the Beards' *Basic History of the United States*:

> To the discussions men and women had flocked on horseback, in farm wagons, in carriages and on foot. . . . They followed the arguments of the debaters and weighed the clashing opinions soberly, with due recognition of their sig-

0272-1732/89/0600-0009$01.00 © 1989 IEEE

> nificance. At the same time newspapers had carried far and wide full reports of the debates; and citizens all the way from Maine to California could make up their minds on the merits of the arguments and the plans for meeting the impending crisis.

What Frances Moore Lappe has tried to do, in the absence of substantive discussions of issues by present-day public figures, is to concoct a Lincoln-Douglas debate for our own time. Of course, since she has provided both sides of this debate, and since she clearly identifies with one of them, the obvious question of impartiality arises. However, what struck me in reading this book was how well matched the two sides were. There are no knockout punches in this contest.

The issues in this debate are of a general nature, but they go to the heart of the problem that Robert McIvor addresses from his perspective as a semiconductor executive. Are economic health and social justice related? Are they mutually exclusive, mutually reinforcing, or something in between? How do you define and measure either of these conditions? How do they relate to freedom?

As there are few one-liners or sound bites in this dialogue, I'm not going to try to give you any highlights. Go get the book and read it.

Miscellany

***Dictionary of Computer Terms*, 3rd ed.** (Webster's New World, New York, 1988, 426 pp.; $6.95)

I don't know why I bother reviewing books like this, but some things irk me so much that I can't resist. None of the definitions in this book struck me as egregiously bad. On the other hand, none that I can recall seeing as I browsed through it had any depth or gave me any new insights. Some were, to put it kindly, a little flaky. For example:

> *Ada.* It is named for Ada Augusta Lovelace, the first female programmer....
>
> *Add-in.* Refers to a component that can be added to a circuit board already installed in a computer. . . .
>
> *Blanking.* On a display screen, not displaying a character or leaving a space.
>
> *Chain.* (1) Linking of records by means of pointers in such a way that all records are connected, the last record pointing to the first. . . .
>
> *Queue.* Queues are nothing more than the waiting lines that have become an accepted and often frustrating part of modern life. . . .

There is a sample program occurring in the definition of *Basic*. It contains the following line:

170 REM ** P1 - PRINCIPLE FOR THE PRESENT MONTH **

Will someone please send the Webster's New World folks a dictionary of English.

HyperDictionary, Philip J. Brown (Van Nostrand Reinhold, New York, 1988, 220 pp.; $19.95)

After waxing enthusiastic about Hypercard, Brown tells us in his preface: "The only thing missing has been a complete and comprehensive guide to the Hypertalk language. . . . Here it is."

Well, no it isn't. This book is a creditable and useful piece of work, but it has a few shortcomings. For example, Brown shrugs off the possibility of interfacing with compiled code as "not something most users will want to do," and fails to deal with it at all, let alone completely and comprehensively.

Another mistake he makes is the common one of failing to distinguish between parameters and arguments. His definition of *argument* is: "Used in some computing books for mean parameter."

Starting from this point, he now tries to explain how to define functions, and the reader is left wondering whether the function's parameters are global variables, like Basic subroutine arguments, or are limited in scope to the body of the function.

Another place where Brown leaves the reader wondering is in his attempts to describe arithmetic. He says several times that Hypercard does arithmetic by using the SANE package, and if he would just leave it at that he'd be on safe ground. Unfortunately, he doesn't. For example, here are excerpts from his entry, *number storage*:

> Hypercard stores everything as strings of characters, even numbers. . . . When it has to perform numerical operations, Hypercard first converts the strings into its internal representation of numbers. If the result is . . . stored in a variable, it will retain the internal representation, and so its full accuracy. The internal representation defaults to 6 places of decimals because the default value of numberFormat is '0.######.'

Either Hypercard's behavior is bizarre, or Brown has confused internal representation with printing format. Similarly, he identifies specific decimal representations of *e* and pi as being built into Hypercard, although one assumes that Hypercard would identify these constants to SANE by name and use its values for them.

I don't want to keep carping about these fine points. This is not a bad book, but it doesn't live up to its promise. The Hypertalk documentation void remains unfilled. This is a job for Apple, and Apple ought to do it.

***Digital Filters*, 3rd ed.,** R.W. Hamming (Prentice Hall, Englewood Cliffs, N.J., 1989, 300 pp.; $48)

In his preface Hamming says:

> Digital filtering includes the processes of smoothing, predicting, differentiating, integrating, separation of signals, and removal of noise from a signal. Thus many people who do such things are actually using digital filters without realizing that they are; being unacquainted with the theory, they neither understand what they have done nor the possibilities of what they might have done. Computer people very often find themselves involved in filtering signals when they have had no appropriate training at all. Their needs are especially catered to in this book.

One way Hamming caters to the needs of "computer people" is by assuming only calculus and a little statistics, which he reviews. He assumes no background knowledge of electrical engineering. Of course, he develops additional mathematics as he needs it—the subject is basically mathematical. He presents the mathematical arguments clearly. The formulas are competently typeset and clearly displayed. The figures are clear, readable, and well integrated with the text.

In his acknowledgments Hamming thanks the Naval Postgraduate School (Monterey, Calif.) "for providing an atmosphere suitable for thinking deeply about the problems of teaching." He seems to have made good use of that opportunity. If you need to learn about digital signal processing (doesn't everyone?), and you've been put off by the formidable appearance of the books you've seen on the subject, get this one.

For the Record, Carol Pladsen and Ralph Warner (Nolo Press, Berkeley, Calif., 1989, 295 pp. plus diskette; $49.95)

For the Record is a book and a program (PC and Macintosh versions are available). The program is the focal item, but the book is of more importance and higher quality than most program manuals. In fact, the program and book are an outgrowth of an earlier book that provided forms to support a manual personal record-keeping system.

Nolo Press specializes in self-help law books. In this case the legal angle is estate planning, and a major objective of the record keeping automated by the program is to allow your affairs to be understood and managed in the event of your incapacity or death. Of course, well-kept records may certainly be useful to you even if you are not incapacitated.

The great virtue of the record-keeping program is its exhaustive nature. You begin in and continually return to a menu of 27 main categories of records. Selecting any of these opens a further menu of subcategories, and selecting a subcategory takes you to a screen designed for that subcategory. Each screen marks the start of an entry for that subcategory. Some subcategories have single-screen entries, while others continue over several screens. You can open as many entries as you like in each subcategory and move through a linked list of them using forward and backward buttons at the bottom of the screen. Any category can be assigned a password and locked, a precaution which, the authors admit, can be circumvented by a dedicated hacker.

Each entry is designed with fixed fields into which you enter the specified information—serial numbers, dates, descriptions, locations, or whatever. The fields have fixed upper limits on their sizes, but additional information can be appended by opening a notes window for the entry.

The largest part of the book, called "Legal and Practical Information about Your Records," parallels the main menu of the program. The authors discuss each category in a general way, with minimal direct reference to the entries and fields implemented in the program. Another part of the book is a 35-page treatise entitled "Estate Planning Basics," which is even less directly related to the program. This kind of information is available from many sources. Reviewing it doesn't really fall within my area of expertise, but it seemed clear and well written to me.

The remaining parts of the book—no more than 20 percent in all—are "How to Use *For the Record*," the actual manual for the program, which comes in two flavors—PC and Macintosh; and "Reference," a glossary, subcategory index, and extremely brief troubleshooting guide.

I spent several hours entering my own personal records and am far from finished. I have a variety of small quibbles about the design of the user interface and the editing and publishing of the book, but they're not worth mentioning. I like this program, and I think I'll finish compiling my personal records someday, if I can find the time.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 177 **Medium** 178 **High** 179

Computer Projects in Japan

Ken Sakamura
University of Tokyo

To learn what is going on in another country is often difficult. You may come across some articles with a few statistics or some comments about the country in which you are interested. But putting these pieces together to form a comprehensive picture of the scene in a foreign country is no easy task. I hope the articles assembled in this issue of *IEEE Micro* help readers acquaint themselves with the Japanese computer scene.

Since the articles in this issue can cover only a tiny portion of the computer activities, I summarize here very briefly three of Japan's ongoing computer projects, all major projects of particular interest to *Micro* readers. The Fifth Generation Computer project develops artificial intelligence systems, the Sigma project strives to increase the software productivity of the Japanese computer software industry, and the TRON project, the one in which I am deeply involved, aims at establishing a computer system architecture.

The Fifth Generation Computer and Sigma projects have strong financial backing from Japanese government agencies. The TRON project, on the other hand, receives support from 130 commercial organizations that include European, American, and Japanese companies.

The 10-year Fifth Generation Computer project is now into its eighth year of operation. Last year (1988) saw the demonstration of Multi-PSI, a cluster of 64 PSI-II sequential inference machines. Using these inference machines, project designers developed KL1, a parallel language; PIMOS, a multiprocessor operating system; and PIM (Parallel Inference Machine), which runs Prolog rapidly and will eventually contain 100 processors. In the last three years of the project, the stated goal is to build a prototype of PIM that contains 1,000 processors.

The Sigma project, started in 1985 and extending to 1989, aims at making a standard software development environment available to computer software developers. Sigma-OS (a variant of Unix), a group of software productivity tools called the Sigma tools, and Sigma workstations form the basis of this environment. Plans call for setting up a centralized depository of topics on software productivity tools known as the Sigma center. This center will disseminate the latest information to Sigma workstations connected to it, thus promoting software productivity by providing the latest tools and recycling reusable components. In 1990 we expect to see Sigma-OS version 1 and the full use of Sigma workstations and tools.

0272-1732/89/0600-0012$01.00 © 1989 IEEE

The TRON project's goal is to produce an HFDS, or Highly Functionally Distributed System, in which a very large number of computer objects are connected. The number, on the order of millions and more, far exceeds the number of processors in existing computer networks. The network is heterogeneous. Currently, project participants work at designing the components of the computer architecture to make the HFDS a reality.

The readers of *Micro* are aware that I introduced articles about the TRON project in 1987 and in 1988. This latest update on the project should mention that six companies now produce the VLSI CPU family of chips based on the TRON specification. The Gmicro/200 and TX1 CPUs are commercially available, and samples of the floating-point processor unit, cache controller, and DMA controller to be used with the CPUs have been released. Various sources, including US companies such as Microtec Research, Inc. in Santa Clara and Ready Systems Corp. in Sunnyvale, California, offer the software tools to support software development for the CPU family. Designers are also working on a general-purpose system bus called Tobus/Toxbus.

The industrial version of the TRON operating system, ITRON, now includes a smaller specification called Micro-ITRON that is targeted to single-chip CPU application. Micro-ITRON will be used in high-end home appliances and will be important in developing electronics goods that connect to the HFDS environment in the future. Also in design is the ITRON2 (an interim name) operating system for the VLSI CPU in the TRON project.

Designers working with BTRON, the business version of TRON, have produced some prototypes for software development. BTRON incorporates multimedia capability, as you can see from one of the articles in this issue. Samples of such machines will become commercially available in Japan through special outlets such as third-party software publishers or system houses interested in building new BTRON applications.

The network operating system tying these pieces together is called MTRON, or Macro TRON. MTRON researchers have formed and activated TRON computer housing, computer building, and urban development projects. The projects include construction companies and furniture manufacturers in their membership. The inclusion of these participants will ensure that the HFDS can really be built on an experimental basis. Their activities will bring new insights to future TRON design activities.

The articles from Japan in this issue contain a discussion of a data-driven VLSI processor for consumer electronics developed by Mitsubishi Electric, Sharp, and the Osaka University team. Hitachi's article concerns a numerical processor developed for the TRON project that should also be of interest to workstation designers. The BTRON/286 article describes the first implementation of the BTRON architecture on 286-based computers.

I hope the articles in this issue give you some idea of the current development activities in Japan.

Ken Sakamura is an associate professor in the Department of Information Science at the University of Tokyo. He initiated the TRON project in 1984. Under his leadership, several universities and over 100 manufacturers now participate in the project to help build computers in the 1990's. In addition to his involvement with TRON, Sakamura chairs several committees of the Japan Electronics Industry Development Association and the Information Processing Society of Japan. He has written numerous technical papers and books and received the BS, ME, and PhD degrees in electrical engineering from Keio University in Yokohama. He is a member of the IEEE and the IEEE Computer Society.

Questions concerning this article can be directed to Ken Sakamura, Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 150 **Medium** 151 **High** 152

FEATURE

An Overview of the BTRON/286 Specification

While individuals may find the 32-/64-bit TRON Project's systems too costly, this 16-bit version should be more affordable and fill business data and graphics needs.

Ken Sakamura
University of Tokyo

Yoshiaki Kushiki
Matsushita Electric Industrial Co., Ltd.

Kazuhiro Oda
Toshiba Corporation

BTRON, or Business TRON, is a set of operating system specifications for workstations and personal computers that employ bitmapped displays. The primary goals set for BTRON call for the realization of easy operation through a standardized human-machine interface (HMI) and the assurance of data compatibility across different applications and machines. Second, BTRON establishes a method for exchanging data and control across applications. Third, BTRON realizes a document management system in which text and graphics can be mixed freely and layouts handled with ease. And fourth, it offers standard support of English, Japanese, and other languages, providing a wide variety of fonts.

The BTRON specification incorporates an innovative information management model known as the *real/virtual object model*.[1] Other new concepts include BTRON's use of tags called *fusen* to define relations between applications and data. HMI specifications extend from keyboard and pointing-device design to the movement of objects on the screen and basic editors for both text and graphics. The standardized HMI, data compatibility, and other aspects take form in a consistent design approach followed throughout the entire TRON-project architecture.

The BTRON specification achieves maximum efficiency when it is implemented on a family of processors designed as part of the TRON project: the VLSI CPU. The clearly hierarchical operating system design also assures ease of implementation when the nucleus is realized on other processors. Here we explain some details about this nucleus and the extended nucleus of the BTRON specification. We concentrate on the BTRON/286 specification, which is suitable for currently available 16-bit microprocessors. (See the BTRON/286 hardware and video image handling boxes on pp. 16 and 17.) Additional features, not found in the BTRON/286, will be added to BTRON/TRON VLSI CPU specification. Refer to Sakamura[1-3] for a general overview of BTRON/286 plans.

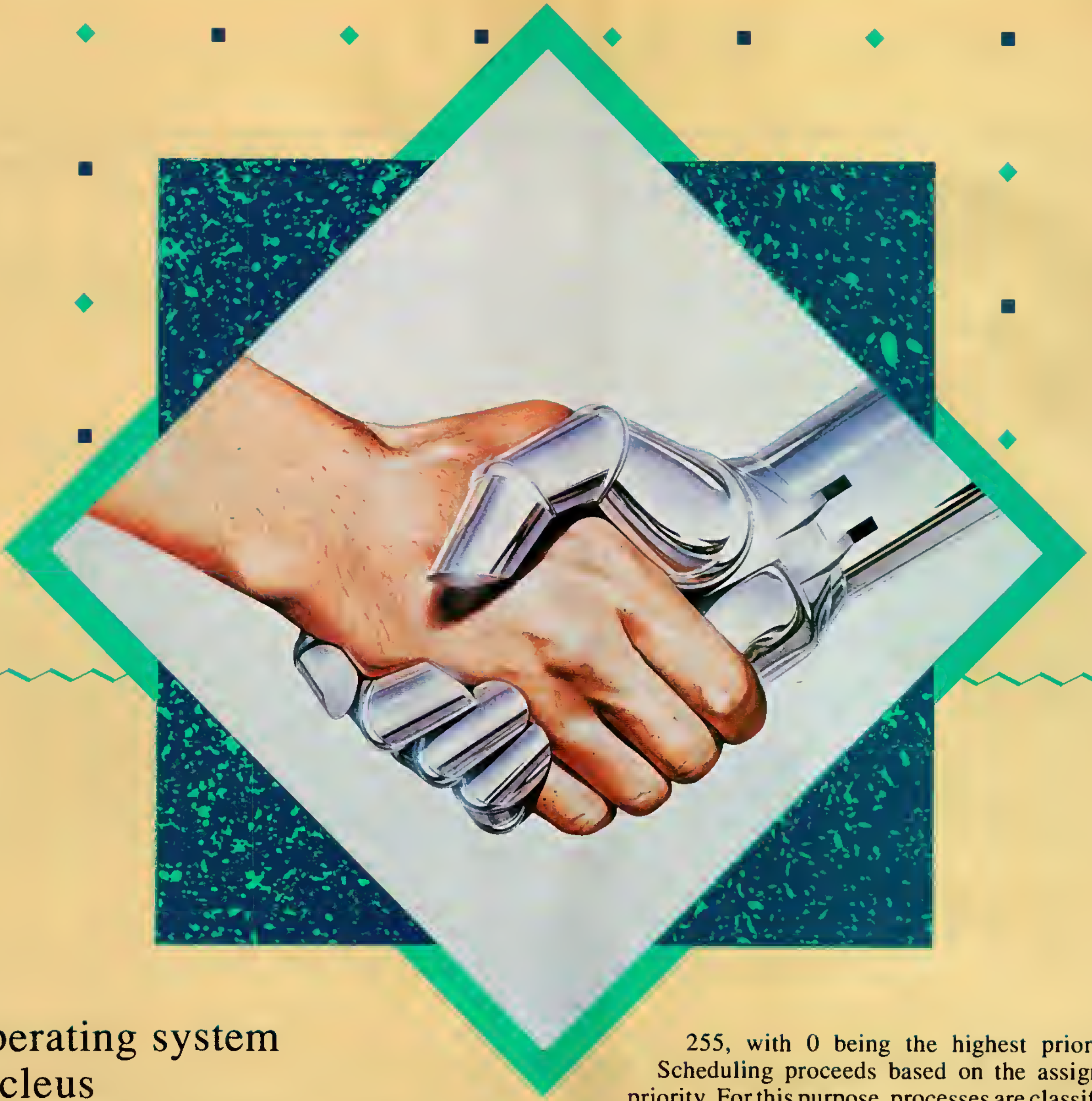
Software configuration

The BTRON specification defines a single-user, multiprocess operating system designed for a superior human-machine interface. It consists of the following three broad levels:

- the operating system nucleus,
- the extended nucleus, and
- system applications.

Figure 1 on p. 16 outlines the BTRON software configuration.

0272-1732/89/0600-0014$01.00 © 1989 IEEE



Operating system nucleus

The operating system nucleus provides basic services such as process management, interprocess communication, synchronization, and a systemwide naming space. Let's look at some of the features.

Process management. Processes are discrete units of the overall processing involved in program execution. A process ID (>0), given to a process when it is created, distinguishes the different processes. Processes have four basic types of status (see Figure 2 on p. 17):

- *Nonexistent*. The process has not been created.
- *Ready*. The process is executable and waiting to be dispatched.
- *Running*. The process is executing.
- *Waiting*. The process is waiting for a message, the passage of an indicated amount of time, input/output, and so on.

Processes go through the following transitions from one status to another, as a result of system calls or scheduling (dispatching, preempting).

Process priority and scheduling. When a process is created, it receives a priority numbered between 0 and 255, with 0 being the highest priority. Scheduling proceeds based on the assigned priority. For this purpose, processes are classified into three groups: absolute priority (priority between 0 and 127), round-robin 1 (priority between 128 and 191), and round-robin 2 (priority between 192 and 255). Scheduling takes place as follows:

1) If there are processes in the absolute priority group on ready status, the process with the highest priority changes to running status and executes. Otherwise, scheduling moves to step 2.

2) If there are processes in round-robin group 1 on ready status, scheduling assumes relative priority (explained later). The selected process (not necessarily the one with the highest priority) changes to running status and executes. Otherwise, scheduling moves to step 3.

3) If there are processes in round-robin group 2 on ready status, scheduling assumes relative priority. The selected process (not necessarily the one with the highest priority) changes to running status and executes. Otherwise, scheduling starts over from step 1 above.

Relative priority is a dynamic priority that is calculated using the statically assigned priority, CPU time usage of the process, and frequency with which the process has been scheduled. Relative priority keeps a

The BTRON/286 Hardware

The BTRON/286 is currently the only TRON Project implementation that can run on hardware within reach of an individual user. The specification of the hardware appears in Table A and is comparable to an IBM PC AT. Although a hard disk is not mandatory, designers highly recommend it. Since the BTRON/286 executes the protect mode of the 80286 CPU, 16 megabytes of address space can be accessed. Figure A is a photograph of the computer that runs the BTRON/286.

Table A. Specifications.

Unit	Description
CPU	Intel 80286
Clock	8 MHz
Main memory	3 megabytes
Floppy disk	3.5 inch, 1 megabyte × 2
Hard disk	20 megabytes (optional)
Display resolution	640 × 400
Display colors	16 out of a 4,096 palette
Keyboard	The TRON keyboard

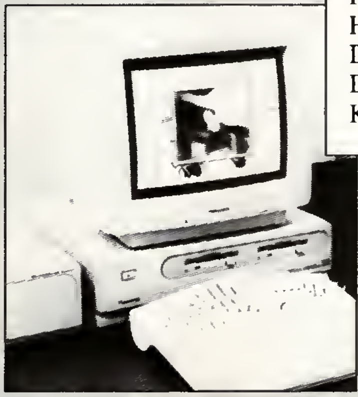


Figure A. An 80286-based computer runs the BTRON/286, an operating system based on the BTRON specification. The machine contains a special Video Processor Unit that makes it possible to superimpose video images from external sources on the screen.

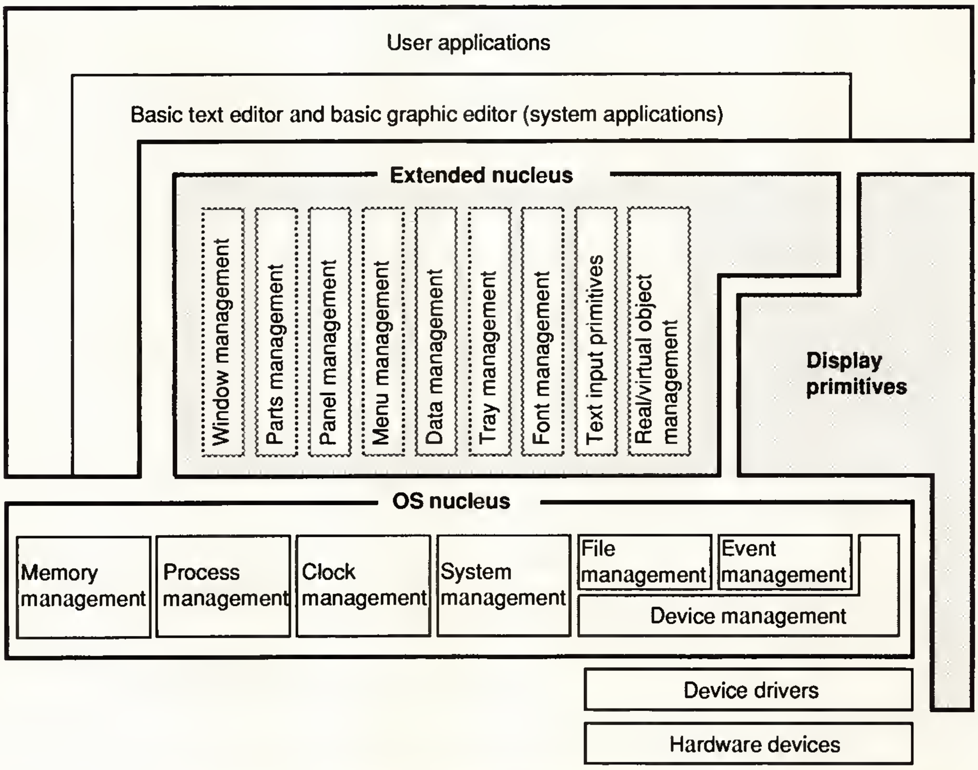


Figure 1. BTRON software configuration.

Video Image Handling on the BTRON/286 Implementation

The BTRON/286 hardware uses an optional Video Processor Unit to handle video image data. The VPU permits external video signals to be displayed on the screen of a personal computer, then digitized and stored into internal frame memory. Users can enlarge or shrink the image as well as repeat small images (tiling). Also, the digitized images can be shown superimposed on the computer screen. Other operations such as inversion and logical operations on the digitized images are available. With this VPU users can manipulate animated images taken from a videodisc or videocassette recorder system.

The extended nucleus of the BTRON/286 operating system contains a built-in software module called the video manager. This module isolates the application programs from the details of the video hardware and offers a set of logical interface functions to access the video images. This isolation makes it very easy to develop application programs for handling many digitized images. Figure B is an example of an application that uses the video manager on the BTRON/286.

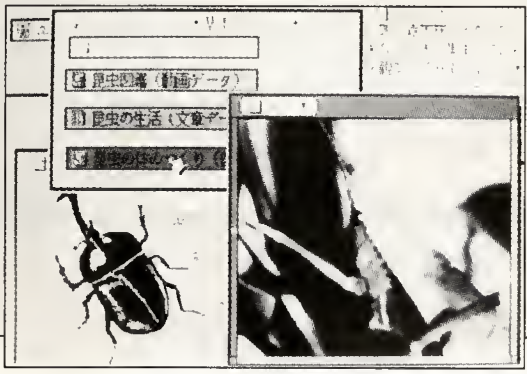


Figure B. Video, graphics, and text windows opened simultaneously on the BTRON/286 screen.

low-priority process from waiting forever.

Normally, system processes and real-time processes belong to the absolute priority group, while ordinary application processes belong to the other two groups. A process in the absolute priority group (priority between 0 and 127) can lock or unlock itself and can become resident in memory by locking the memory space of the process in memory. Such a process is said to be on lock status. The priority of a process once it has been created can be changed only within the same group; it cannot be shifted to a different priority group.

Message communication among processes. Each process contains a message queue in which messages are communicated among processes. A message moves to the message queue belonging to a process according to the destination indicated in its process ID. See Figure 3. The source is likewise identified by means of the process ID in the sending process. A message queue has a FIFO (first in, first out) structure, so that messages always enter the queue in the order in which they are sent. If the message queue at the destination is full when a message is sent, the sending process receives an error code that tells it to wait until space becomes available. Certain types of messages can be received selectively by specifying their types.

Synchronization and control among processes. Counting semaphores act as mechanisms for synchronization among processes and for mutually exclusive usage of shared resources. A semaphore ID assigned at

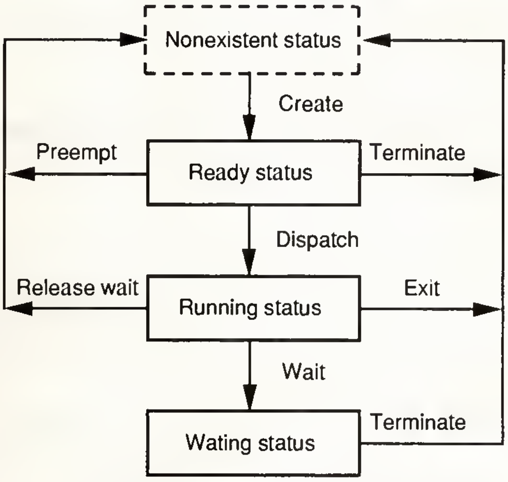


Figure 2. Basic status transitions of processes.

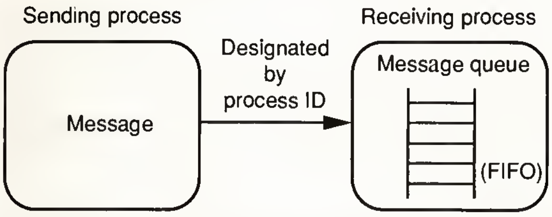


Figure 3. Message communication among processes.

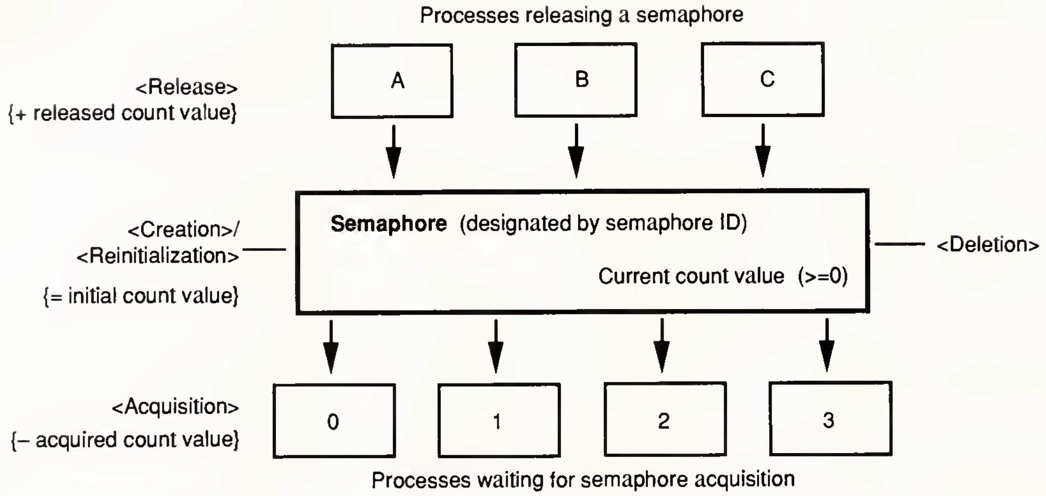


Figure 4. A semaphore operation.

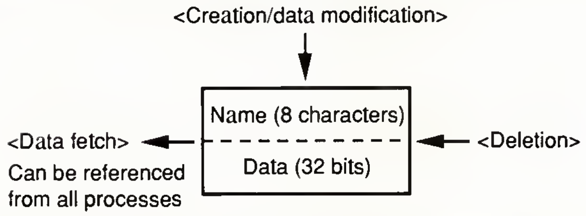


Figure 5. A global name data operation.

the time of creation identifies the dynamically created semaphores. Once a semaphore has been created, it can be used by every process. Figure 4 shows the semaphore operation.

Global name data management. Global name data can be defined as data with a globally visible name. Global name data can be shared among processes so they can exchange a small amount of data with high speed. Such data contain freely chosen names up to eight characters long by which all processes can reference and modify the data, as seen in Figure 5. The two kinds of global name data include one that continues to exist regardless of the existence of the process creating or modifying it and that must explicitly be deleted when no longer needed. (In BTRON/286, the modification of the global data is treated as if it were a creation of new data.) A second kind is automatically deleted when the process creating or modifying it is terminated. Either kind can be designated when creating or modifying global name data. Global name data mainly allow certain data to be referenced by any process.

Memory management. The BTRON specification provides for management of local and shared memory.

Local memory. This type of memory can be used in only one process and cannot be accessed directly by other processes. Local memory is created automatically at the same time a process is created and is released automatically along with the process at the time the process terminates. Application processes can acquire and use memory blocks of the required size from local memory. The acquired memory block contains continuous (logical) addresses, and the start (logical) address is returned to the application.

Shared memory. This type of memory can be used by all processes for data that are shared by several processes, for data used by the extended nucleus, and for other purposes. A number of memory pools make up a shared-memory area, though at system start-up, only the system memory pool exists. Application processes acquire shared-memory blocks of the specified size from the specified memory pool, a special memory pool consisting of the entire available system memory. Processes dynamically create ordinary memory pools that are identified by a memory pool ID given at the time of creation. Creation of a memory pool amounts to allocating part of the system memory pool and making it a separate, discrete memory pool. See Figure 6.

File management. The file structure adopted in the BTRON specification maps the real/virtual object model for the actual hardware. BTRON/286 users will manipulate the data based on this model, but the underlying data structure is implemented as files, as in conventional systems. Note that

• Files are structured as record streams; that is, they consist of ordered series of variable-length records.

• A random network of reference relationships exists among files based on links (virtual objects) included in files. (No directory exists as in conventional file systems.)

• These links (virtual objects) access files directly.

• Access is controlled via a user access level number (0-15) that is assigned to each user and a file access level number (0-15) that is assigned to each file.

Files and links. As just mentioned, a *file* is a stream of ordered, variable-length records that corresponds to a real object. A *link*, corresponding to a virtual object, points to files and serves as a clue for file reference. A link can be embedded in any file as a record, and more than one link can point to the same file. In this way an overall network of reference relationships is defined. Links indirectly reference files, and as a result the name of a file does not have an absolute meaning to uniquely identify a file. Rather, it functions as one search key. Any file name of up to 20 characters can be assigned, and the same name can be assigned to more than one file if desired. See Figure 7.

File structure. Every file system has one root file. By tracing the links included in the root file, users can reach all files in the file system, as shown in Figure 8. In the real/virtual object model, a root file corresponds to a device's real object. When a file system is created, it acquires a file system name up to 20 characters long and a device location name. The system and the users need the file system name, and the name of the root file, for absolute identification of the file system. The device location name, also up to 20 characters in length, indicates the physical device in which the file system is stored.

Attaching the file system. At system start-up, no file system exists. Before one can be used, an attachment operation as shown in Figure 9 must take place, whereby the name of the logical device in which the file system exists and the attaching name are specified. Detaching an already attached file system is done by specifying the name of the logical device to the detaching command. As a result, one cannot access a file via links that reference the files in the detached file system. The links are said to be on detached status. In the real/virtual object model, a link on detached status corresponds to a shadow object.

File ID. A file ID is a unique 16-bit number assigned to all files in the file system. The file ID of the root file in a file system is always 0.

Direct and indirect links. A fixed link, a link stored in a file as one record, can reference only files in the same file system. A link file is a special file to control

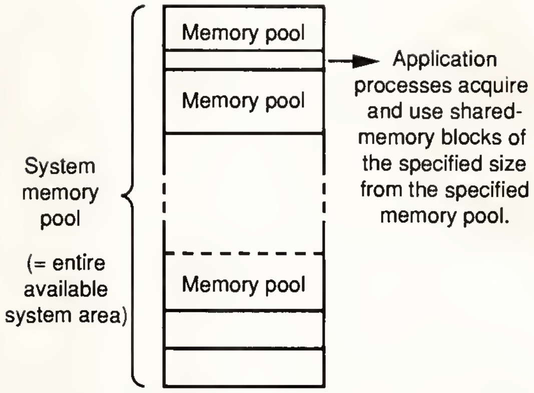


Figure 6. System memory pool.

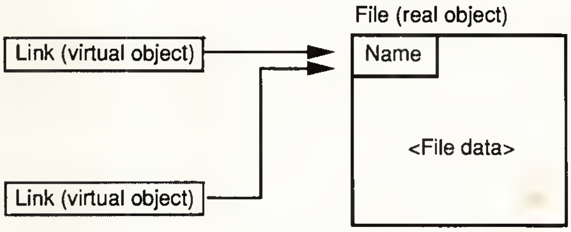


Figure 7. Files and links.

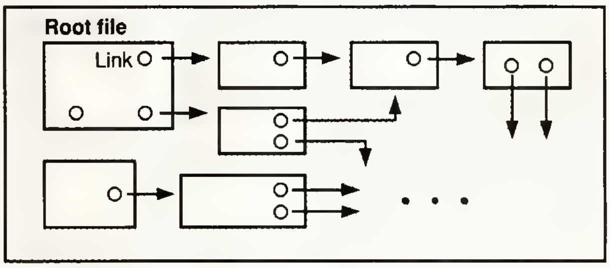


Figure 8. File system structure.

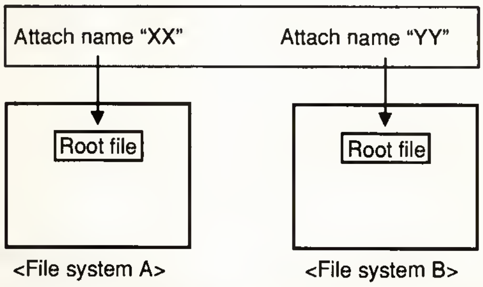


Figure 9. A file system attachment.

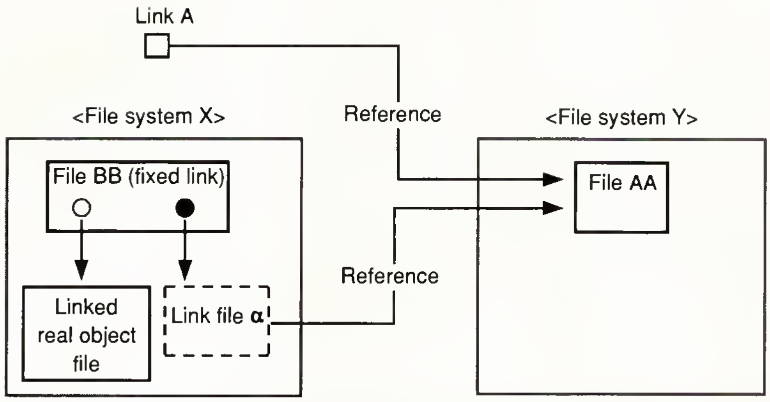


Figure 10. An indirect link and a link file. To store link A in file BB, first create link file α then store indirect link A' to α in file BB.

a reference to different file systems for configuring an indirect link. References to files in different file systems must be made via a link file in the originating file system. Links to link files are called indirect links because they add another level of indirection. As can be seen in Figure 10, references to files in the same file system can be made via a direct link stored in a file. An indirect link is a fixed link to a link file; a direct link is a fixed link to an ordinary file.

Event management. This capability uniformly treats keyboard and pointing-device operations as *events* by which users can interact with the computer. See Figure 11. These operations are recorded as one event after another in the systemwide event queue. Applications fetch these events one after another from the event queue, with the corresponding action being carried out in an event-driven form. This arrangement is based on the rule that, at a given point in time, only the process engaged in dialogue uses the event management capability to fetch events. The BTRON/286 specification defines the following types of events:

- button down,
- button up,
- key down,
- key up,
- automatic repeat key,
- device,
- null, and
- application events 1 to 8.

Each event is associated with a bit mask, which identifies the event a process has fetched.

Device management. Device management functions include an application interface for the uniform handling of various devices, device-driver registration and management for various devices, and a device-driver interface. See Figure 12. The file management and event management capabilities in the operating system nucleus perform the actual device operations. A logical device name is a unique character string registered in the system and consisting of class, unit, and subunit elements. A *class* name indicates the type of device, such as *hd* for hard disk, *fd* for flexible disk, and so on. A *unit* distinguishes individual devices when a number of units of the same class exist. Normally, a single letter of the alphabet, starting from *a*, identifies units, for example, hda0, hda1 for hard disk hda. When one unit is logically subdivided, a *subunit* distinguishes each part using numbers from 0 to 254 assigned in order.

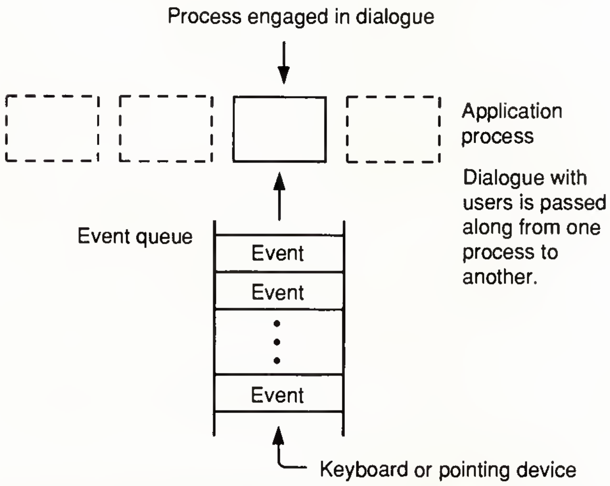


Figure 11. Event management. At a given time, only the process engaged in dialogue fetches an event, making use of the event-management capability.

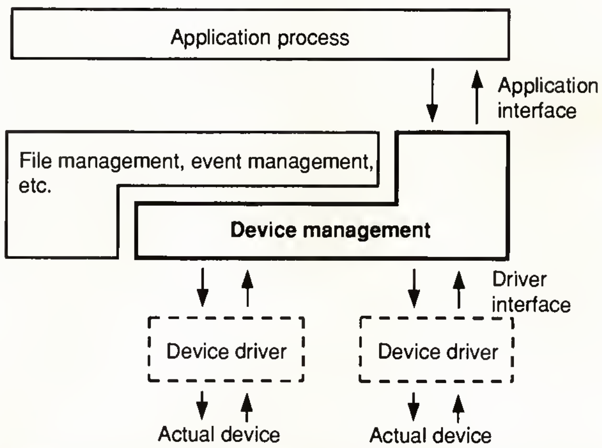


Figure 12. Device management.

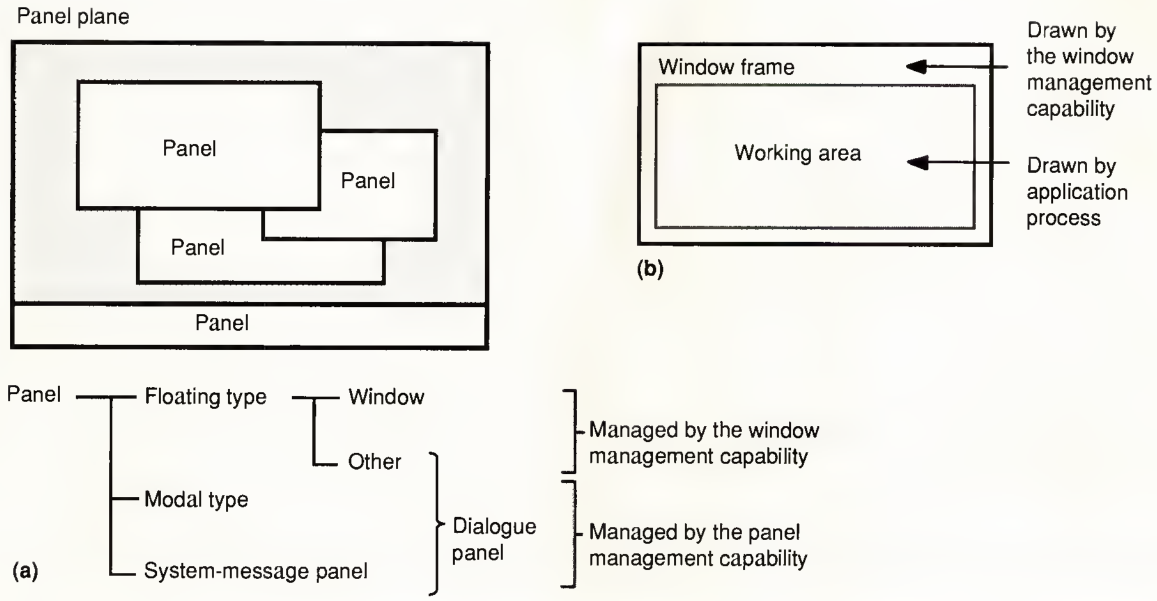


Figure 13. Panel (a) and window (b) management. The panel plane is the operation environment space on the display screen.

Extended nucleus

As shown earlier in Figure 1, the software contains many functions. Here, we explain three special features of the BTRON operating system:

- panel management,
- real and virtual object management, and
- TAD (the TRON Application Databus).

Panel management. The BTRON specification designates a rectangular region on the (virtual/real) display screen as a *panel.* A panel can be placed in an overlapping manner any place on the screen and displayed via programs. When a hidden panel is exposed (say, if a panel that obscures another panel is moved), the newly exposed area in the panel is redrawn to show the data that should be visible on the screen. The display of each panel is independent of what is shown in other panels.

When different panels appear on the screen, only one panel can accept a user's keyboard input. The user must click the button on the pointing device (a penlike accessory) to change the status of the panels so that a different panel can accept input. The BTRON specification makes it possible to realize a panel (windowing) system with a minimum of hardware and memory. A basic assumption is that no special hardware is supported.

Panel types. Panels can be classified into modal, floating, and system-message categories, as shown in Figure 13. Users call up *modal* panels on the screen via application programs; these panels can't be moved once they are displayed. Modal panels are always exposed unless hidden by other modal panels. When a modal panel is displayed, it is the only panel with which a user can interact.

A user can relocate *floating* panels by dragging the pointing device over the screen. When a floating panel can accept input, the user can choose to click the pointing device and access another panel so it will accept input. A floating panel that shows the content of a real object and allows users to interact is called a *window* in BTRON terminology. Usually, a window shows a part of a real object, and the user scrolls the screen to take a look at different parts of the object. Floating and modal panels that are not windows are called *dialogue* panels. Dialogue panels often offer the user an interface for setting application parameters.

The *system-message* panel is a long, narrow panel that appears at the bottom of the display screen. Only one system-message panel can appear on a screen. It holds system messages and is never hidden.

Windows can be extended to occupy the full screen (save, of course, the system-message panel) and are said to be in full-screen mode. The user can easily change a window so that it occupies either a small region or a full screen.

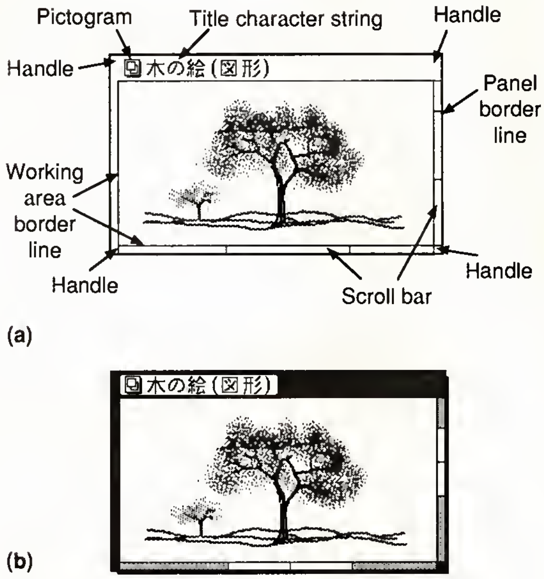


Figure 14. Standard window shape and title bar display (a) and a window in the input enable status (b).

Window display. Figure 14 shows the standard window shape and standard title bar display.

Window operations. Please recall that a window is a type of panel. There are five window operations:

- *Creating a window.* The user executes an application using a menu and clicks twice on a pictogram so that the selected virtual object is opened in the window.
- *Switching input enable status.* The user either clicks on the inside of the window with the pointing-device button or selects the real object name from the display management menu. (This allows the window to move to input enable status and become the first front window.)
- *Moving a window.* The user drags the pointing device over any part of the window frame other than a handle or scroll bars.
- *Resizing a window.* The user drags a handle inside the window frame to choose an arbitrary window size or a full-screen window. For a full-screen window, the user either selects the full-screen mode from the menu or clicks twice on the handle. Users can switch between the full-screen mode and normal (original) size easily.
- *Erasing a window:* The user either selects Quit from the menu or clicks twice on a pictogram.

Pointer shape. The screen displays a pointer shape (in the form of a hand for easy understanding; see Figure 15), which indicates the active operation. When neither a pointing-device button, menu button, nor Command key is pressed, the pointer shape depends on the pointing-device position and automatically changes to a shape that indicates the operation possible in that position. (On a standard TRON digitizing pen, the pointing-device button is located at the tip of the pen and the menu button in the middle of its side. The Command key appears on the TRON keyboard.)

Real/virtual object management. These functions let the user display and manipulate real and virtual objects, as well as register and delete application programs. They also allow those application programs making use of real/virtual object management to display and manipulate real and virtual objects. In addition, all application programs fundamentally are executed via real/virtual object management. The real/virtual object management functions include:

- various functions for display and manipulation of virtual objects,
- various functions for display and manipulation of fusen,
- application program management (registration/deletion/ execution),
- menu setting,
- file system-mounting management, and
- other application functions.

Virtual object operations. Listed here are the operations possible with regard to virtual objects. Users activate these operations directly on the screen either by using a pointing device or by selecting an item from a virtual object operation menu. The operations are

- selection,
- removing a disk,
- copy or movement,
- resizing,
- opening,
- closing,
- modification of a real object's name,
- modification of a relation,
- creation of a new revision,
- display of virtual/real object management information,
- opening as a window,
- modification of display attributes, and
- registration or deletion of an application.

Physical implementation. These objects are mapped to physical data that consist of many types of records. For example, a real object can consist of records that contain application programs, program-specific data, and TAD (explained next) format data that can be shared between applications and many systems. Each record of these objects contains subdivisions called segments.

TAD (TRON Application Databus). The BTRON specification provides a uniform data format called TAD, which permits the ready exchange of data among different applications. A TAD record makes up a part of the real objects in BTRON.

From the standpoint of assuring data compatibility between applications, TAD can be seen to have these four features:

- guaranteed compatibility of basic data (text, figure, and sound) across applications;
- variable-length segment format for easy data-exchange processing;
- the capability to hold application-dependent data; and
- a realization of the real/virtual object model in exchanged data, allowing external data referencing.

The fusen, handling application-specific data in TAD. TAD specifications are designed to guarantee data compatibility across applications at a minimum level. The TAD format is structured of text and figures as basic data, based on the notion that text and graphics are two kinds of data that can be handled by any application. However, if we define data format for text and figures in a very restricted way, many programs may suffer from the lack of flexibility and loss of efficiency. To avoid this problem, TAD permits an application-specific data format and clearly separates the application-specific data and application-independent data.

Application-specific data, such as a database index or a special binary data structure, are stored as *fusen* (a Japanese word for a small piece of paper used as a memo pad). Fusen is a generic name for application-specific segments. Although the content of a fusen is application-specific, its external format follows a certain rule. Among others, the application name that can interpret such fusen is stored in a defined format at the beginning of each such fusen segment. So when an application encounters a fusen segment, it can safely ignore the segment since it knows the length of the segment. In a sense, BTRON standardizes the way nonstandard data is stored.

Fusen can be stored like virtual objects inside a real object and are displayed as rectangular regions on the display. However, unlike virtual objects, each fusen itself holds some data internally. (A virtual object is a link to other real objects.)

Usually the user can change the content of a fusen by showing it on the screen and then invoking the associated application by clicking to have a dialogue panel appear. The user actually changes fusen by interacting through the dialogue panel.

Fusen can be classified broadly into the following types:

- *Functional.* This type of fusen holds data necessary for invoking application programs. If a functional fusen is embedded in a real object, the fusen can specify the application that works on the real object.
- *System setting.* This type of fusen modifies system settings such as the volume of a beeping sound, which repeats automatically whenever a key is held down for a specific length of time.
- *Adjective.* This fusen is embedded in the text or figure record of a real object. An application will use the adjective fusen to interpret data with added detail. For example, font information or format information for a character is stored as adjective fusen in a TAD format data. The basic text editor, which is available on any BTRON-based system, can interpret the adjective fusen and act accordingly.

The following manipulations can be performed on fusen:

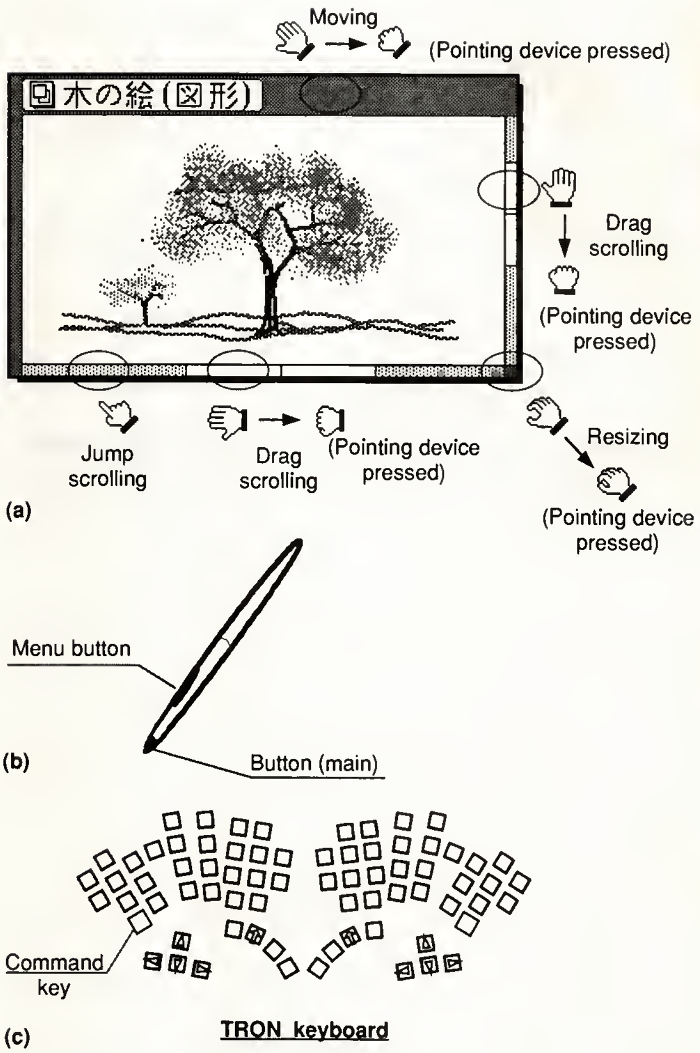


Figure 15. The pointer shape interacting with the screen (a), the pointing device (b), and the TRON keyboard (c).

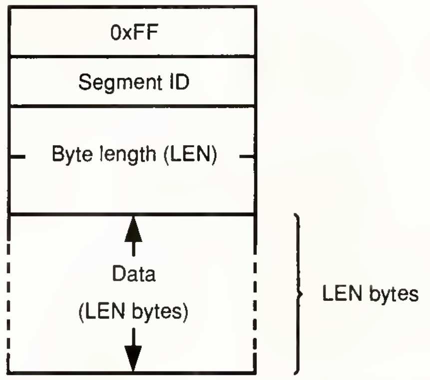


Figure 16. The structure of a variable-length segment.

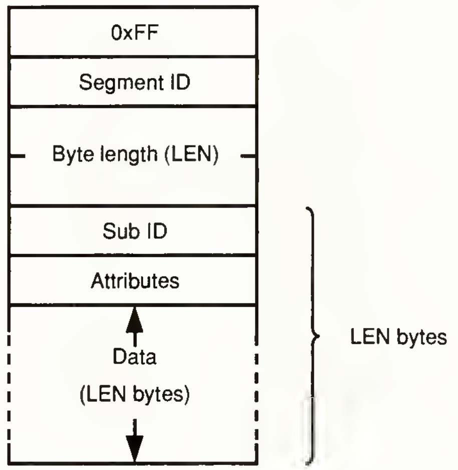


Figure 17. The structure of a text fusen segment and a figure-drawing segment.

• selection (same as with a virtual object);

• copy or movement (same as with a virtual object);

• modification of the name (same as modifying a real object name);

• opening or setting parameters (same as opening a virtual object as a panel; the corresponding application is executed, and normally a dialogue panel is displayed); and

• modification of display attributes (same as modification of display attributes of a virtual object, except that the setting items are virtual object subsets).

As an example, a spreadsheet or database application adopting a tabular organization consists of cells. The attributes of each cell and information on table structures are stored as adjective fusen in TAD. The text data can be handled by any application, whereas only certain applications can process the adjective fusen.

TAD guarantees data exchange, at least of the data in each cell of the table-based application, as character strings. In this way various applications can make use of the table data at the most basic level, that of character-string data.

Variable-length segments. TAD consists of structured data called variable-length segments or variable-length data having the structure 0xFF + segment ID + data length + parameters, as shown in Figure 16. The 0xFF code at the beginning of the segment equates with the escape code in the TRON code system. 0xFF indicates the start of the segment, and the segment ID indicates the type of fusen. The data length gives the data size of the parameters that follow. The fusen segments and figure fusen segments contain a sub ID giving further information on the type of segment. In this case the structure appears as shown in Figure 17.

We gain two advantages from classifying types of fusen and adopting a structure that includes segment ID and byte-length fields. Applications unable to process a segment can skip it, based on data-length information, and go on to process the next data. This means that applications are not required to support every segment, and there is less burden on the system developers. Secondly, there is no need to maintain a large fixed-length table of formatting information.

Holding application-dependent data. The application-dependent data that are not covered by the TAD specifications can be stored in three possible ways: in the application program record, in the adjective fusen record, and in the text-application fusen or figure-application fusen.

The basic mechanism adopted in BTRON to associate the application-specific data and the application program itself is to embed a segment that holds the application ID. This ID designates the application that can interpret the data. The choice of the method to use in embedding the application ID is not specified at this moment. We believe the necessary underlying mechanisms support the idea of storing the application-specific data in TAD to promote the maximum portability.

Virtual object segments. TAD realizes the real/virtual object model by means of link records and virtual object segments. A link record holds link information to a real object, while a virtual object segment holds information for visually displaying a virtual object.

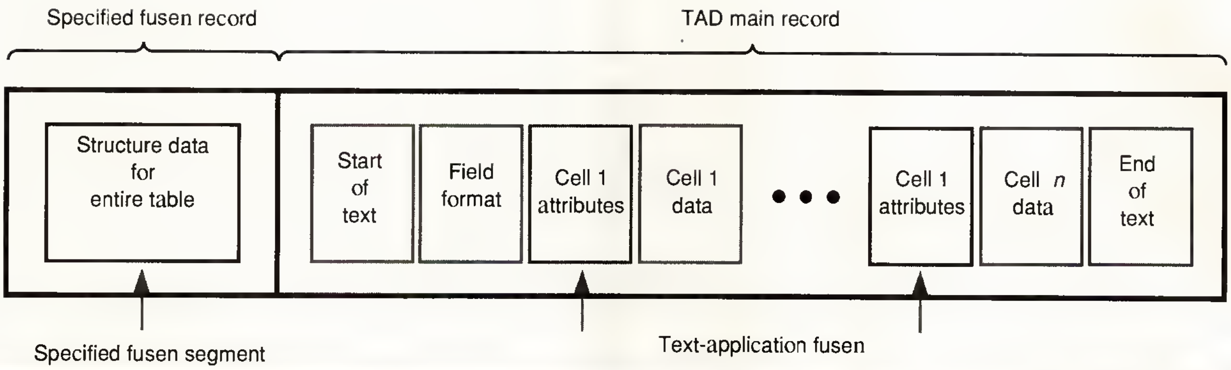


Figure 18. Data representation based on the real/virtual object model.

When data contain a virtual object, TAD saves the virtual object segment by embedding it in the text or figure data. The position of the segment shows the relative position of the virtual object in the data, and the order of appearance of real images defines the correspondence between link records and virtual object segments. Figure 18 shows how data are represented when two virtual objects exist in the same set of text data.

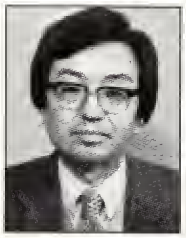
Other standard formats allow data in other files to be referenced, but TAD offers the advantages of referring to the network-type file and of preserving virtual information.

BTRON/286 implements the BTRON human-machine interface on an 80286-based hardware system. Please note that the BTRON human-machine interface can be implemented on different hardware platforms and can be implemented in many ways. Currently, work is proceeding to develop a BTRON human-machine interface that runs on the TRON VLSI CPU. In this implementation, designers emphasize distributed operating system functionality. We hope we can report on the next development in a year or two.

References

1. K. Sakamura, "BTRON, The Business-Oriented Operating System," *IEEE Micro*, Vol. 7, No. 2, Apr. 1987, pp. 53-65.
2. K. Sakamura, ed., "TRON Project 1988: Open Architecture Computer Systems," *Proc. Fifth TRON Project Symp.*, Springer-Verlag, Tokyo, 1988.
3. *BTRON/286 Specification*, The TRON Association, Tokyo, Dec. 1988.

Yoshiaki Kushiki



Kazuhiro Oda

Ken Sakamura's biography, picture, and address appear on p. 13.

Yoshiaki Kushiki currently works in Matsushita Electric Industrial Co., Ltd.'s Information Systems Research Laboratory. He has been engaged in the research and development of system architecture, such as operating systems, databases, the human-machine interface, and artificial intelligence. He received the BE degree in electronics engineering from Kyoto University in Kyoto and is a member of the IEEE.

Kazuhiro Oda is a chief specialist in the Personal Computer Design Department at the Ome Works of Toshiba Corporation. He is now a member of the BTRON and CTRON technical committees of the TRON Association. He has been engaged in software design and development of large and medium-scale computers, small business computers, distributed processors, and word processors. Oda received the BE degree from Kyushu University in Fukuoka.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 153 **Medium** 154 **High** 155

FEATURE

A Floating-Point VLSI Chip for the TRON Architecture:

An Architecture for Reliable Numerical Programming

Preliminary evaluations of this FPU show a good combination of precision and performance. It supports IEEE floating-point arithmetic and the ANSI C draft proposal.

Shumpei Kawasaki
Mitsuru Watabe
Shigeki Morinaga

Hitachi Ltd.

The TRON project[1] advocates a network system consisting of heterogeneous computing nodes called intelligent objects that have a wide range of functions. Figure 1 illustrates how a standard network connects the intelligent objects so that they can communicate with each other. The TRON architecture also specifies a processing element to be used as a component of the intelligent objects for the VLSI CPU,[2] a standard instruction set for a central processing unit. Multiple vendors will provide the Gmicro microprocessor family,[3] a series of VLSI (very large scale integration) chips conforming to the architecture. Among these microprocessors are the Gmicro/100, Gmicro/200, and Gmicro/300.

While the TRON architecture for the VLSI CPU handles integer, bit-field, string, list-structure, and bitmapped data, it excludes the floating-point data intended to be handled by the coprocessor or library. As some intelligent objects will undoubtedly handle real number data, floating-point instructions are urgently needed in the architecture for the VLSI CPU. With this in mind, we designed the Gmicro/FPU to provide floating-point instructions for both the Gmicro/200 and the Gmicro/300. The VLSI CPU architecture defines 23 coprocessor instructions, some of which are designed to be used in the floating-point instructions.

Since the TRON project is meant to provide information infrastructure for various layers of society, the reliability of its parts is the key issue. The intelligent objects can participate in navigation, construction, manufacturing, chemical-plant control, air-traffic control, and even exploration of the universe. Therefore, their reliability can affect the lives of individuals and the public in general. As these applications massively utilize floating-point numbers, the numerical integrity of the Gmicro/FPU potentially could become a social issue. Thus, though performance is an important criterion in the development of the Gmicro/FPU, accurate data are even more important.[4]

Background

A floating-point number is an attempt to express real number data in digital information. A conventional floating-point number consists of a sign bit, exponent field, and mantissa field as seen in Figure 2.

0272-1732/89/0600-0026$01.00

The value of a floating-point number is determined by:

$$\text{value} = (-1)^s * 2^{e-\text{bias}} * m$$

where s is the sign bit, e is the exponent, and m is the mantissa. Bias is a constant that enables the floating-point number to express a value larger or smaller than one. A wider exponent field would increase the range of the value expressed by the format. A wider mantissa field would increase its resolution.

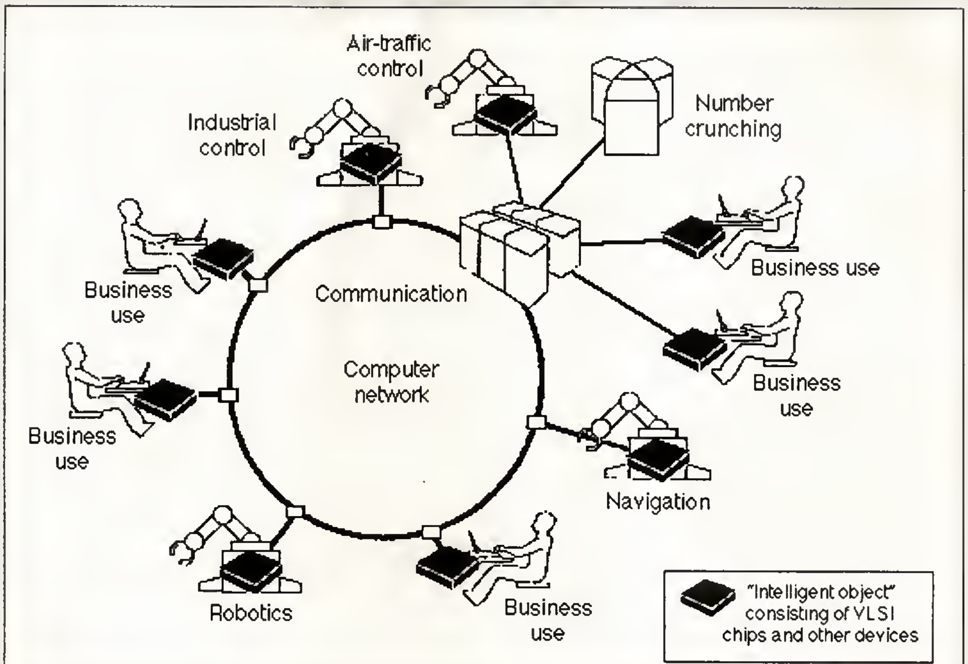


Figure 1. The standardized network structure linking "intelligent objects" advocated by the TRON project. Details of the construction can be found in Sakamura.[5] Standardized operating systems are proposed.

Floating-point hardware and software. Prior to 1960 computers used instructions to handle integers and programmers wrote floating-point software libraries utilizing these instructions. The programmers found it difficult to write a floating-point library that executed efficiently and yet was economical in memory usage. Mainframe manufacturers introduced special floating-point arithmetic instructions in the 1960s, drastically improving the execution speed and economy of memory. In the 1970s minicomputer vendors also started to supply floating-point instructions on high-end products. Unfortunately, each manufacturer choose different floating-point data formats. Exchanging the floating-point data and floating-point software between two different machines presented major difficulties.

To resolve the difficulties, in the late 1970s the Institute of Electrical and Electronics Engineers proposed a floating-point standard (now called the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*[6]). Microprocessor manufacturers have unanimously accepted this standard. It explicitly defines the essential hardware features of floating-point arithmetic, including the floating-point data format, semantics of operation, conditional branch mechanisms, and exception handling. All conforming systems give the same result for each operation.

Writing floating-point software in a high-level language, or HLL, increases the software's reliability, portability, and maintainability. However, when written in the current HLLs,[7] numerical software often suffers a decrease in reliability. It is impossible for a programmer to determine the exact floating-point operations that will be executed for a particular source-level construction of an HLL. The floating-point operations generated from an HLL source code remain unpredictable in the eyes of the programmers. Some of the mysteries often encountered in HLL numerical programming include the following:

Sign	Exponent	Mantissa

Figure 2. The components of a conventional floating-point number.

1) Commutative law ($x + y = y + x$) does not hold in many HLL compilers.

2) In Fortran one can test two variables for inequality at one point in a program and find that they are not equal, and then test them again later to find that they are equal.

3) A floating-point program often gives different results before and after a code optimization by an HLL compiler.

To eliminate the unpredictability and ambiguities of floating-point arithmetic in the C language, the American National Standards Institute released a draft proposal for the C language.[7] C when implemented according to this draft proposal makes the exact floating-point operations visible to the programmer.

Summary of Floating-Point Standards

One industry standard and one standard proposal define the boundary conditions of future floating-point hardware implementations. The purpose of both of them is the portability of floating-point software among various systems. One calls for more floating-point software to be written in a high-level language.

The *ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic* explicitly defines essential functions of floating-point arithmetic hardware and software. However, it defines the functions as floating-point operations without regard to other software support such as that in high-level languages. The standard sets

- data formats (32-bit single, 64-bit double, and 80-bit extended-double precisions);
- six standard arithmetic operations (addition, subtraction, multiplication, division, square root, and remainder), requiring a precise solution for each;
- branches on floating-point conditions;
- binary and decimal conversion of floating-point data,
- rounding methods; and
- exception traps and handling methods.

The *Draft Proposed American National Standard for Information Systems—Programming Language C* determines the exact floating-point operations that will be executed for a particular source-level construction. The programmer can estimate what floating-point operations are generated just from the C source program. The draft proposal sets

- precisely defined rules for expression evaluation. In the type promotion rule the binary operators (+, –, *, /) must calculate the result in the data format of the operand(s) with the highest precision. Examples of the type promotion rule include:

  <long double> + < double> –> <long double>
  <float> * <double> –> <double>
  <double> – <double> –> <double>

- strict rules to eliminate optimization resulting in the loss of exactness of operations,
- no change of operation precedence, and
- 21 well-defined mathematical functions.

ANSI is developing the draft proposal to "provide an unambiguous and machine-independent definition of the language C."[7] The standard lets a programmer determine the exact floating-point operations that will be executed by adjusting the source-level construction of C. Most of the mysterious phenomena seen in HLL floating-point programming can be eliminated by explicit rules for the precision of intermediate values in expression in the compiled code and rigorously defined, standard mathematical library functions. Most HLL compilers capriciously choose the precision for intermediate expression in the compiled code, which explains most of the mysterious phenomena described earlier. A programmer can now write numerical software without fearing the hidden "features" of an HLL.

Though the draft proposal sets no specific requirements on floating-point hardware, conventional floating-point units are not suited for efficient execution of a system based upon the draft proposal. Thus designers must give the floating-point hardware more dexterity in controlling the precision of the intermediate values to handle the exact operation demanded by the ANSI draft proposal. The Gmicro/FPU contains an architectural feature that adjusts to the draft proposal without impairing performance. See the box for a summary of the IEEE standard and the ANSI draft proposal documents.

The precision issue. The IEEE floating-point document standardizes the precision of basic mathematical operations such as add, subtract, multiply, divide, square root, et cetera. No precision requirement for the elementary functions is included; it is left up to the implementer. (Elementary functions are frequently used in scientific arithmetic and include trigonometric, hyperbolic, exponential, and logarithmic functions.)

When floating-point hardware is used in critical applications such as navigation, the precision of an elementary function matters a great deal. A fraction of error in a trigonometric function can lead you many miles away from your destination. Hardware designers cannot pass this problem to the software designers, since they are in a position to determine any trade-off between performance and precision. Since the software designers must compete with each other about the performance of the total system, they can hardly improve the precision at the cost of performance. On the other hand, the hardware designers are more likely to be the ones to choose the bit length of the arithmetic unit and the constants. They are also the ones to have the options in implementing special hardware without introducing deterioration of the performance.

Floating-point units

Most microprocessor manufacturers prefer to integrate the floating-point arithmetic functions on a sepa-

rate chip from the CPU known as a floating-point unit. Familiar FPU examples include Intel's i80387 designed for use with the i80386 CPU and Motorola's MC68882 designed to accompany the MC68020 CPU. There are two reasons for producing a CPU and FPU separately. The first is the system's requirement: Floating-point functions are not used in all systems. The second is the nature of VLSI production: The yield Y of a VLSI chip is expressed by

$$Y = a * e^{-D * S}$$

where D is the density of defects, S is the die area, and a is a coefficient factor common to all technology. The technology determines the density of defect D. For high-end, state-of-the-art microprocessors, the yield is very sensitive to small changes in the die area. Thus, implementing the entire CPU function and floating-point arithmetic calculations on one die would cause the yield of the chip to drop substantially, resulting in a more-costly chip.

Coprocessor interface. An instruction extension chip such as an FPU designed to accompany a VLSI CPU is often called a coprocessor. To reduce the number of interchip interconnections, designers build an autonomous control system into the extension chip, making it a cooperating processor rather than an extension of the CPU's logic. The instructions extended by the coprocessors are called coprocessor instructions. When a VLSI CPU encounters a coprocessor instruction, it generates a series of handshake procedures on the coprocessor chip. These procedures, called coprocessor protocols, perform the transfer of the instruction information, data, and state, while leaving the execution of the instruction to the coprocessor. The CPU normally does not "know" the content of coprocessor instruction operation, leaving the detailed definition of coprocessor instructions to a later time when the coprocessor is actually implemented.

The TRON architecture for the VLSI CPU includes 23 coprocessor instructions, which cover all probable interactions with coprocessors to be defined in the future. Since the coprocessor protocol is not part of the architecture and its implementation is left to the manufacturers, the protocol can be defined so that it is optimum to the specific technology in which the chip is fabricated.

The Gmicro VLSI CPUs such as the Gmicro/200 and Gmicro/300 include coprocessor instructions. Programmers see the FPU instructions the same way they see other CPU instructions. By connecting the Gmicro FPU, we add 50 floating-point-related instructions and 19 floating-point-related registers to the Gmicro instruction set, the register resource. Vendors producing the Gmicro family of VLSI chips have defined a coprocessor protocol common to the Gmicro family.

The coprocessor protocol between the CPU and FPU determines much of an FPU's performance. The speed of bare arithmetic hardware, such as the arithmetic logic unit or the multiplier, does not become the performance determinant so long as the chip is not equipped with a faster coprocessor protocol. Manufacturers have implemented various kinds of handshake methods between the CPU and the extension. Some coprocessors scan the instruction stream, identify their own instructions, and activate themselves. Other coprocessors are mapped onto a special memory space so the CPUs can access them as slave processors. The protocol of the Gmicro/FPU and Gmicro VLSI CPUs are the latter type except that some special interface pins are added to decrease the number of bus cycles necessary to complete the coprocessor protocol. The number of bus cycles is a major determinant of the maximum performance of the FPU, since the arithmetic operations can now be performed in very few machine cycles.

New computation algorithms. With advances in technology, vocabularies of the VLSI architecture gradually evolve, and the cost of each numerical operation changes. As a result designers discard traditional algorithms in favor of more appropriate ones such as the Cordic (COordinate Rotation Digital Computer) algorithm for calculating a wide range of elementary functions. Cordic's implementation with a set of adders, shifters, and read-only memory allows most of today's VLSI FPUs to use the silicon area to best advantage.

Instances of an algorithm inspired by a new hardware technology can be found in the Gmicro/FPU. Designers introduced a parallel multiplier consisting of a matrix of carry-save-adders, which multiplies two operands in two or three machine cycles. Previously, when implemented in a microinstruction loop, multiplication took 30 to 70 machine cycles. The drastic reduction in the computation cost of multiplications makes room for new algorithms for IEEE square-root and elementary functions. The new algorithm is potentially faster than any known algorithm when fast multiplication is available.

Requirements

The design objectives for the Gmicro/FPU floating-point unit were set to:

- fully conform to the IEEE floating-point standard,
- provide adequate architectural support for the ANSI C draft proposal,
- provide an efficient CPU interface as well as high-performance floating-point hardware,
- provide elementary functions with the highest precision, and
- provide an entire set of mathematical library functions as instructions.

Architecture

The Gmicro/FPU can be used as a resource of the Gmicro/200. Figure 3 shows the extended register after the FPU is attached to the Gmicro/200. The FPU contains sixteen 80-bit, floating-point data registers (FR0-FR15) and three control registers: the FMCR (floating-point mode), FSR (floating-point status), and FQR (floating-point quotient). The registers function as follows:

- FR0-FR15 registers store floating-point data;
- the FMCR specifies trap enable/disable and rounding modes;
- the FSR stores condition codes, exception flags, and accrued exception flags; and
- the FQR stores the quotient for modulo and remainder instructions.

Instruction set. The FPU's instruction set and its basic formats appear in Table 1 and Figure 4. All instructions for arithmetic operations have only floating-point registers for destinations. Figure 4 displays the major instruction formats. Designers modeled the basic instruction format after the coprocessor instructions in the TRON architecture for the VLSI CPU. Both source and destination operands have size specifiers, which give the unit an unusual architectural strength. The result is rounded to the precision of the destination operand size, exception signaled (if any), and stored in the floating-point data register.

Figure 5 illustrates how, when variables are assigned in floating-point registers, an ANSI C compiler can generate faster code for the Gmicro/FPU than can be obtained with conventional FPUs. The C-language source code in Figure 5a on p. 32 is cited as an example of expression evaluation. According to the ANSI proposal, the evaluation of the expression $u = t * (x + y) + z$ takes the four steps shown in Figure 5b; in each step, the rounding precision is explicitly specified. Both source and destination operands have size specifiers, which enables the unit to conform to the ANSI C draft proposal even when the values reside in its floating-point registers. The earlier box contains the type of promotion rule that determines this precision.

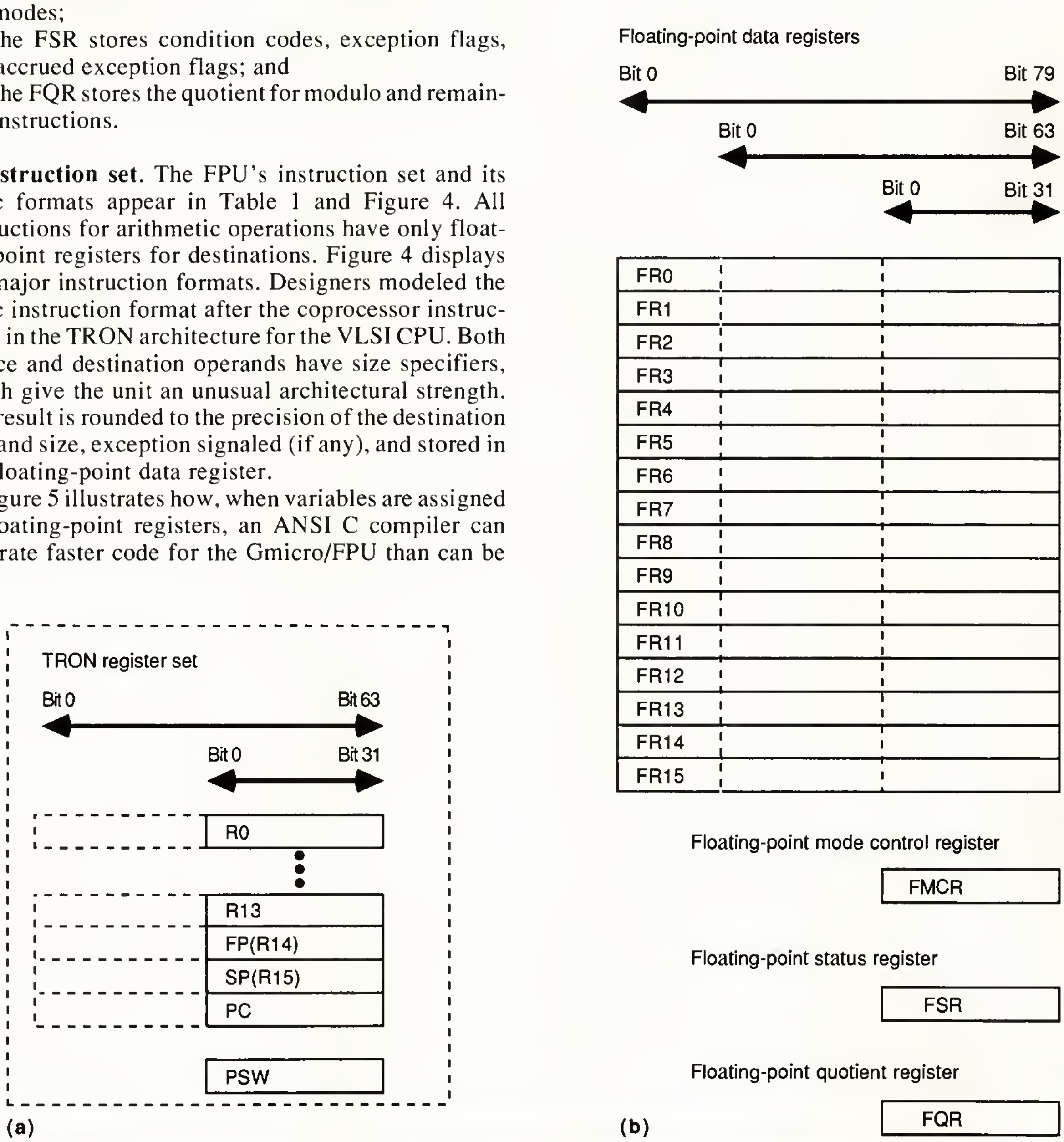


Figure 3. The TRON register set (a) and the extension of the register resource when the FPU is added to the Gmicro chips (b).

Table 1.
Gmicro/FPU instruction set.

Operation	Function
Basic	Addition, subtraction, multiplication, division, remainder, IEEE modulo, square root
General	Absolute value, negation, round to integer, truncate to integer, extract exponent, extract mantissa, generate constant
Elementary functions	Sine, cosine, sine cosine simultaneous calculation, tangent, arcsine, arccosine, arctangent, sine hyperbolic, cosine hyperbolic, tangent hyperbolic, arctangent hyperbolic, exponent base 2, exponent base 10, natural exponent, logarithm base 2, logarithm base 10, natural logarithm
Graphics support	Vector inner product (up to eight dimensions), clipping judgment
Conditional branches	Compare, test, conditional branch
Data format conversion	Load floating-point number, store floating-point number, load signed integer, store signed integer, load unsigned integer, store unsigned integer, load control register, store control register
Operating system related	No operation, internal reset, save internal state, restore internal state, load multiple floating-point numbers, store multiple floating-point numbers

A common tendency in the current floating-point chips is the precision change that occurs when floating-point registers are used in optimization.[4] Most of these chips convert results to the length of the registers—the longest data type a register can hold. Promotion of the floating (32-bit) variable to a long double (80-bit) variable occurs when the destination is a floating-point register. To observe the ANSI proposal's requirement on conventional floating-point chips, we would have to generate the object code shown in Figure 5c, a succession of 10 instructions. The .s and .d suffixes for the assembler mnemonic in this figure indicate single precision and double precision.

In Figure 5d we see the ANSI proposal for object code generation observed on the Gmicro/FPU. Here, only five instructions evaluate the expression, thereby reducing by 50 percent the amount of code necessary with a conventional FPU.

Elementary function instructions. The ANSI C proposal also determines the range of mathematical functions. Mathematical functions defined in the header file <math.h> take double-precision arguments and return double-precision values. The tenet of the function definition is that the domain of the mathematical function must match with the mathematical definition. The Gmicro/FPU instructions conform to this proposal. No software envelope is necessary, and an instruction for an elementary function can simply be generated in line when the functions are called. The complete matching of the domain of the function is ensured. Table 2 on p. 33 lists the correspondence of the instructions and the <math.h> file. Future ANSI expansions are expected to include all operand precisions in elementary functions. This requirement can be met by having the FPU specify the source and destination size in the instruction.

The table also shows the mathematical library function of Fortran 77, which has complex numbers as a data type. Mathematical functions dealing with a complex data type can easily be constructed by using the Gmicro/FPU elementary function instructions. However, this is out of our scope and not shown in the table.

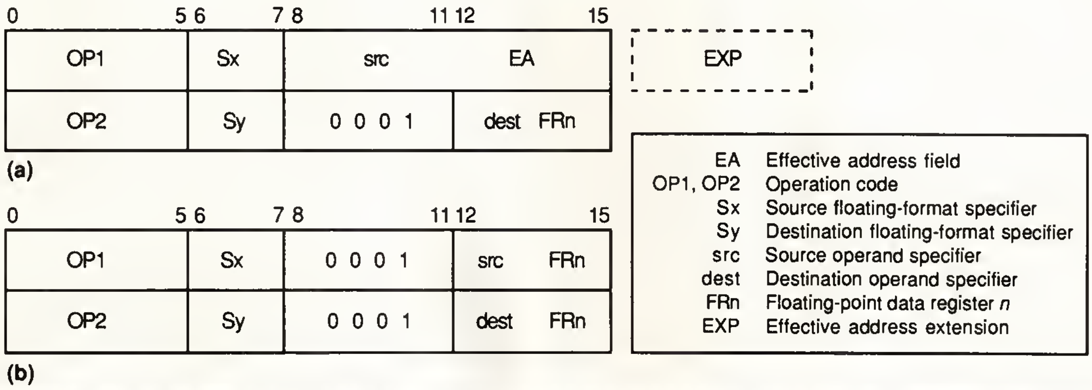


Figure 4. Major instruction formats in the Gmicro/FPU: memory-register (a) and register-register (b). All arithmetic instructions fall under these categories, and only store instructions can store to memory.

```
main()
(
	register float		x, y, z, u;
	register double		t;
	(
		...
		u = t • (x+y)+z;
	}
}
```

(a)

1) $x + y$ is performed to infinite precision and rounded to float or single (32-bit) precision.

2) t • [result of (1)] is performed to infinite precision and rounded to double (64-bit) precision.

3) [result of (2)] + z is performed to infinite precision and rounded to double (64-bit) precision.

4) [result of (3)] is rounded to single precision and stored in u.

(b)

```
sp: stack pointer
fr15 : x
fr14 : z
fr13 : y
fr12 : t
fr11 : u

fmov.s		fr15, fr0	; move x to fr0
fadd.s		fr13, fr0	; add y and store in fr0
fmov.s		fr0,@sp		; store result to round to single
fmov.s		@sp, fr0	; load expression x+y in fr0
fmul.d		fr12 , fr0	; multiply t with content of fr0
fmov.d		fr0, @sp	; round the result to double
fmov.d		@sp, fr0	; load expression t • (x+y) in fr0
fadd.s		fr14, fr0	; add z  to fr0
fmov.s		fr0, @sp	; round the result to single
fmov.s		@sp, fr11	; store the result to u

10 instructions
```

(c)

```
sp : stack pointer
fr15 : x
fr14 : z
fr13 : y
fr12 : t
fr11 : u

fmov	fr15, fr0.s	; move x to fr0 (accumulator)
fadd	fr13, fr0.s	; add y to fr0, round to float
fmul	fr12, fr0.d	; multiply t, round to double
fadd	fr14, fr0.d	; add z, round to double
fmov	fr0, fr11.s	; round fr0 store to u

5 instructions
```

(d)

.s	Single precision
.d	Double precision

Figure 5. Two operand size specifiers on a floating-point instruction conform to the ANSI C draft proposal very simply when the compiler allocates variables in the floating-point registers. C language source code (a); its translation into a floating-point number according to the ANSI standard (b); object code on conventional FPUs (c); and the Gmicro/FPU optimization (d).

Implementation

Here we describe a new, tightly coupled coprocessor protocol for the Gmicro/200-Gmicro/FPU system. One of the design objectives in this protocol is to minimize the CPU-FPU communication overhead to improve the total performance of the FPU. To achieve this goal, we judiciously reduced the amount of CPU, FPU, and memory bus transfers needed for instruction execution. We introduced the following schemes in the protocol:

1) *Some functional redundancies between the CPU and the FPU.* The decoder and the microcode of both the Gmicro/CPU and the Gmicro/FPU store the coprocessor instruction and protocol sequence. Both chips "know" exactly what is to be done once the coprocessor instruction is determined. This capability significantly reduces the bus cycle count over that found in other FPUs, since no coprocessor instruction knowledge must be transferred from the coprocessor to the CPU.

2) *Protocol reduction.* Coprocessor status pins (CPST) reduce the number of bus cycles in a protocol. The Gmicro/CPU can monitor the Gmicro/FPU's status without having to invoke a bus cycle.

3) *Direct data transfer.* The Gmicro/200's bus control circuitry directly transfers data between the memory and the Gmicro coprocessor. Eliminating data transfers between the CPU and the coprocessor normally required for memory-coprocessor data transfer again reduces the number of bus cycles required for the protocol.

4) *CPU write-through cache support.* The Gmicro/FPU supports a write-through cache on future CPUs. Securing the data coherency between the Gmicro/CPU's internal cache and external memory becomes an easy task.

5) *Multiple coprocessor support.* Up to eight coprocessors can logically connect to the CPU. In this way, the CPU can distribute the work load to several coprocessors, making overlapping operations on multiple coprocessors possible.

6) *Fast floating-point conditional branch.* Conditional branch tests on the coprocessor's internal state take place using the coprocessor status lines, thereby making the operation faster.

An example multiple-coprocessor system. Figure 6 depicts a multiple-coprocessor system made up of Gmicro family chips. It contains four coprocessors including an FPU. A common clock serves both the CPU and the coprocessors as the time reference for

Table. 2.
Mathematical libraries and instructions for the Gmicro/FPU.

Instruction	ANSI/C proposal math.h	Fortran 77/ANSI	Instruction	ANSI/C proposal math.h	Fortran 77/ANSI
fabs	fabs	ABS	fsincos,fdiv	—	COTAN
facos	acos	ACOS	fsinh	sinh	SINH
fasin	asin	ASIN	fsqrt	sqrt	SQRT
fatan	atan	ATAN	ftan	tan	TAN
fatan,fdiv	atan2	ATAN2	ftanh	tanh	TANH
fcos	cos	COS	floge	log	LOG
fcosh	cosh	COSH	flog10	log10	LOG10
fexp	exp	EXP	fmod	modf	MOD
fexp2	frexp	—	—	pow	—
fint,fintrz	floor, cell	AINT,ANINT, NINT	fmul	*	*, DPROD
			fneg	-	SIGN
fscale	ldexp	—	fsqrt,fsincos, fatan	—	SQRT
fsin	sin	SIN			

synchronized operations. The synchronized clock enables all of the processors to operate essentially as one processor. Each coprocessor contains input signals called coprocessor identifications, or CPIDs, to uniquely assign an identification for user software. The system initializes the CPID in the reset operation. All processors share the data bus, address bus, and part of the control bus. Notable control signals include bus accesses BAT0-2 and coprocessor status CPST0-2. The CPST0-2 and the BAT0-2 for each coprocessor are OR-wired electrically or logically.

Bus access signals BAT0-2. The Gmicro/CPU uses the bus access signals to inform the Gmicro/FPU of contents on the data bus. The CPU

1) sends an operation's content to the coprocessor,
2) writes the operand data into the coprocessor,
3) reads the operand data from the coprocessor,
4) coordinates the direct data transfer from the coprocessor to the memory,
5) coordinates the direct data transfer from the memory to the coprocessor, and

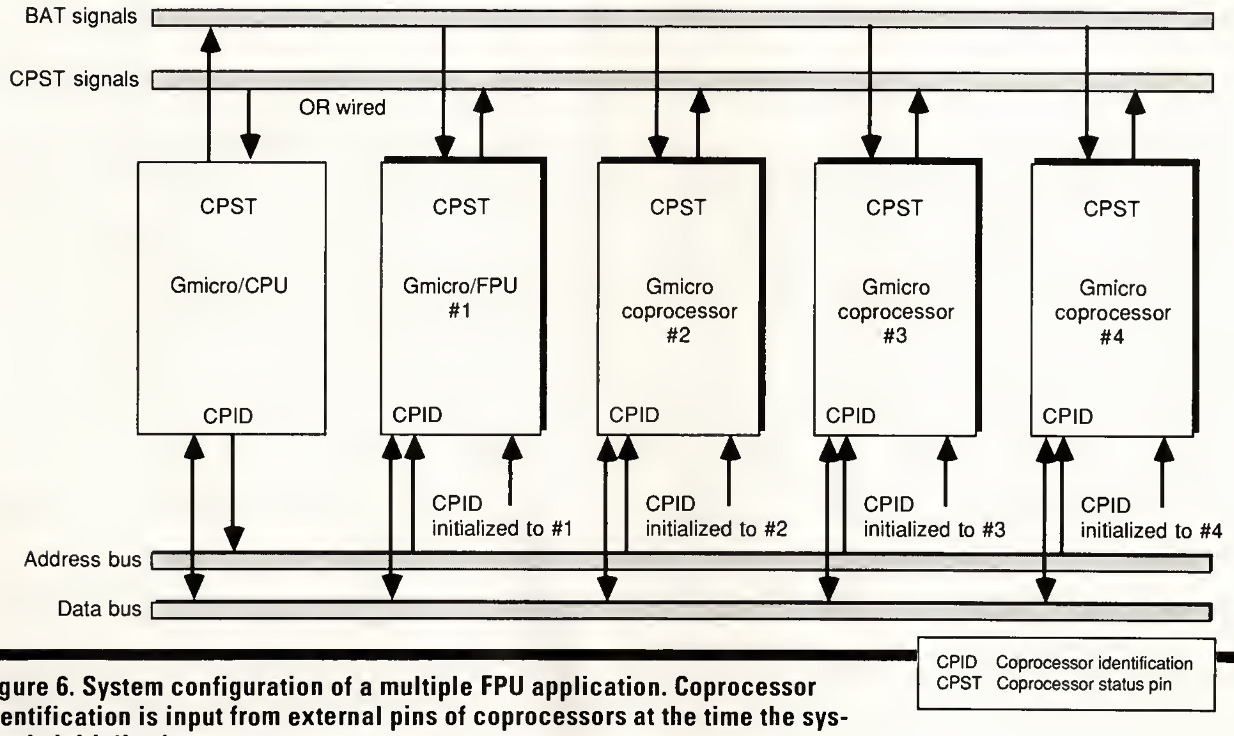


Figure 6. System configuration of a multiple FPU application. Coprocessor identification is input from external pins of coprocessors at the time the system is initialized.

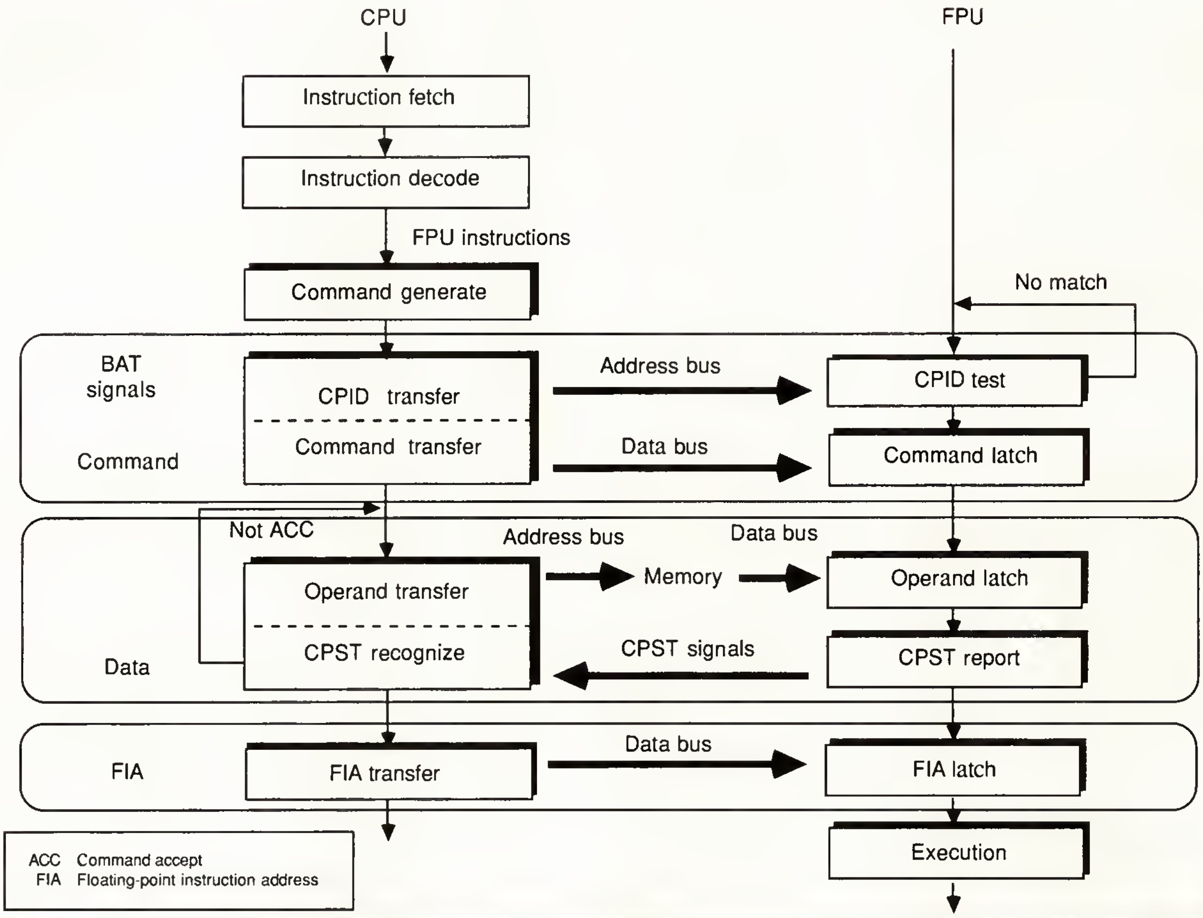


Figure 7. Protocol sequence in the Gmicro/FPU when executing a coprocessor instruction that loads a single-format operand and completes an arithmetic operation.

6) sends the coprocessor instruction address to the coprocessor.

The BAT0-2 signals along with a read/write signal from the CPU determine which one of the above bus access types are currently taking place. From this information, the Gmicro/FPU sends, receives, or processes data on the data bus for each bus access type.

Coprocessor status signals. The CPST0-2 signals report the coprocessor's status to the Gmicro/CPU. The sampling timing of the CPST0-2 signals is determined within the protocol. Normally, these signals are sampled on the second bus cycle in the protocol. The first bus cycle is the operation content transfer. In this case the coprocessor compares its internal state and the operation content to see if the operation can be processed immediately and returns the CPST0-2 signals. The status that a coprocessor could present over the CPST0-2 signals include:

1) *Command acceptance.* The operation required by the CPU is acceptable and immediately processed.

2) *Command error.* The coprocessor does not recognize the operation.

3) *Coprocessor busy.* The coprocessor cannot accept the operation, and therefore the CPU must redo the first two bus cycles in the coprocessor protocol.

4) *Coprocessor exception.* The coprocessor detects an exception in the previous operations, and therefore the CPU must take an exception vector.

5) *Data transfer ready.* The coprocessor is ready to accept or send data operands. This status is reported for the execution of a coprocessor instruction involving multiple operands.

6) *Condition true.* The conditional branch test on the coprocessor state is true.

7) *Condition false.* The conditional branch test on the coprocessor state is false.

Basic types of protocols. A combination of the data, command, and address transfers on the bus cycles and status tests in a certain order form the protocol sequence for a coprocessor instruction. Twelve kinds of protocols perform Gmicro/FPU instructions including internal state save/restore, multiple-register load/store,

and floating-point arithmetic. The floating-point arithmetic operations fall into three groups:

- an operation on the FPU's internal registers only;
- an operation using an operand(s) external to the FPU, CPU register, memory, or CPU cache and storing the result in the FPU's internal register(s); and
- an operation using an operand(s) internal to the FPU and storing the result in the resource external to the FPU, memory, or CPU register.

A coprocessor protocol example. Figure 7 illustrates a coprocessor protocol for a Gmicro/FPU dyadic instruction, an instruction involving two operands to produce the result in an operand. One operand resides in the memory and the other in the Gmicro/FPU's register. This example helps to explain the coprocessor protocol. In executing this coprocessor instruction, the Gmicro/CPU and Gmicro/FPU perform the following activities:

- *Instruction decoded, operation content extracted.* The CPU fetches an instruction and decodes it. If it is a floating-point instruction, the CPU extracts the information needed by the FPU from the coprocessor instruction.
- *Operation content transferred to the FPU along with the coprocessor ID.* The FPU transfers the operation content and command. When this bus cycle completes, three of the address lines indicate the value of the CPID number to which this coprocessor instruction is sent. The coprocessors monitor the three address lines and compare the CPID number on the address lines with their ID numbers. The coprocessor whose CPID matches with the number acknowledges the command and asserts the CPST0-2 lines. Other coprocessors designate their coprocessor status lines as three-state conditions.
- *The CPU coordinates the operand transfer from the memory to the FPU.* The CPU asserts the bus control lines and the address lines and instructs the main memory to put an operand word on the data bus. At this point the FPU asserts the CPST0-2 lines, and the CPU samples them. The FPU receives the operand on a data bus, and the CPU proceeds with the protocol if it detects the command acceptance status. If a coprocessor busy status is detected, the CPU reverts to activity 2. If another coprocessor status is presented, exception actions are taken. CPST0-2 detection and the operand transfer occur simultaneously. If the operand transfer requires more than one bus cycle, either because the size of the operand is larger than 32 bits or because the operand is placed out of alignment with the memory, the bus cycles repeat until the operand transfer completes.
- *The CPU transfers the FPU instruction address to the FPU.* The CPU transfers the floating-point instruction address via the data bus from which the FPU receives the FPU instruction address. This step ensures the availability of the exception information necessary in case an error occurs at a later stage. When the FPU acknowledges the instruction address transfer, the protocol sequence terminates.

Bus-cycle reduction. Figure 8 replicates the actual time chart used for the protocol when the FPU performs a floating-point arithmetic operation of the kind just mentioned: one operand of single-precision data format residing in memory and one operand in floating-

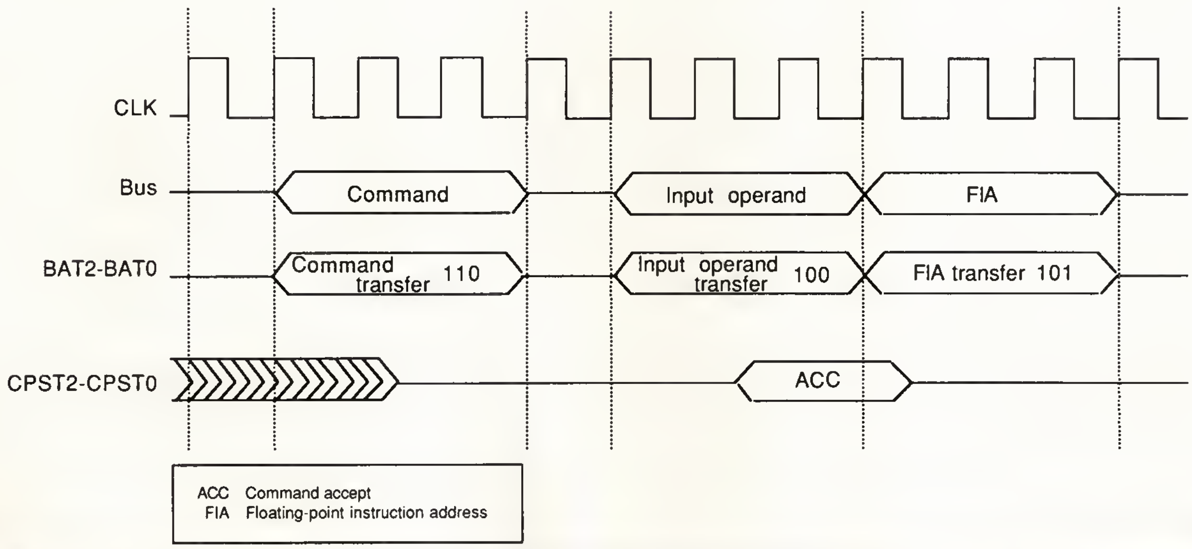


Figure 8. FADD command-cycle timing (memory + FR). A clock cycle equals the machine cycle. The Gmicro/220-Gmicro/FPU system executes this protocol in 10 machine cycles.

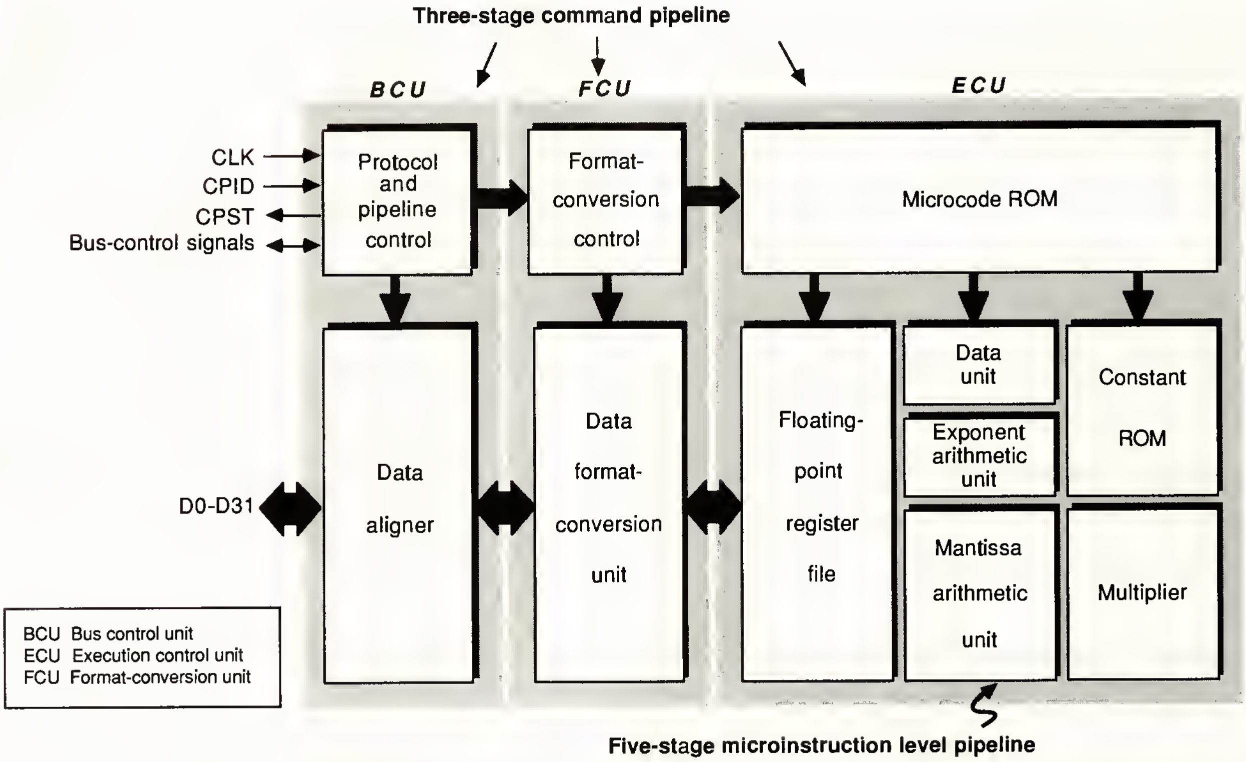


Figure 9. Internal structure of the Gmicro/FPU. The BCU, FCU, and ECU units constitute the active operational blocks, and another piece of logic arbitrates them.

point register with the result stored in a floating-point register. This operation takes 10 machine cycles to complete the handshake.

In this specific protocol the coprocessor status line saves one bus cycle, and direct data transfer between the memory and the Gmicro/FPU saves another bus cycle. Thus we save a total of two bus cycles from the original five—a 40 percent reduction. This amount of reduction is entirely effective since 10 machine cycles are also consumed by the floating-point addition for a single-precision operand. The new protocol scheme matches bandwidths between the high-performance floating-point arithmetic hardware and the protocol.

Other aspects. The FPU can also be used with a CPU without Gmicro coprocessor protocol when it is connected as a peripheral. The CPU emulates the transfers in the protocol sequence of coprocessor operation with the exception that the coprocessor status is read from a special register address. For future Gmicro/CPUs with nonwrite-through data caches, the Gmicro/FPU can retain the data coherencies between the external memory and the CPU cache. When the correct data is in the memory, the CPU can activate the bus cycle from the memory to the FPU. When the correct data is in the CPU's data cache, the CPU can write the data into the FPU. The FPU does not differentiate between the data access methods, carrying out the protocol in the same way wherever the operand originates.

Internal FPU structure. Figure 9 shows the Gmicro/FPU's internal structure. The three main elements are the:

- *Bus control unit (BCU),* which is responsible for handshaking with the main processor. The unit carries out bus cycles, generates the status of the coprocessor, and processes the coprocessor protocol. It also carries out the necessary command/data/status transfers between the main processor and the FPU.
- *Format conversion unit (FCU),* which converts all operands sent from the main processor or the memory to the internal floating-point data format before the actual arithmetic operation starts. In the FPU one internal data format represents all floating-point data. Once the store operation of the floating-point data completes, the data expressed in the internal format are again converted into the external formats, as defined in the IEEE standard.
- *Floating-point execution unit (ECU),* which consists of the microcode ROM and four other units: data type, exponent arithmetic, mantissa arithmetic, and multiplier. The IEEE floating-point data format expresses those values such as infinity, zero, and not-a-

number. The arithmetic operations involving these special values bypass normal sequencing and occur in the data unit. The sign bit also resides in this unit and is manipulated here. The exponent arithmetic unit processes addition/subtraction operations for the exponent calculations. The addition, subtraction, shifting, division, and other necessary operations are performed on the mantissa of floating-point values in the mantissa arithmetic unit. The quarter-size flash multiplier (33 bits × 33 bits) performs high-speed multiplication operations on the mantissa of the floating-point numbers. This multiplier unit is used frequently when calculating square roots, elementary functions, vector inner products, as well as multiplications operations.

Three-stage pipeline. In addition to the coprocessor interface with the reduced bus cycle overhead, the Gmicro/FPU controls a three-stage pipeline. Often the entire protocol processing becomes hidden in a floating-point arithmetic operation. Figure 10 illustrates how the three-stage pipeline control increases the throughput of the FPU by allowing command or data fetches in the BCU, data conversions in the FCU, and floating-point arithmetic operations in the ECU to be performed concurrently.

Excluding the instruction fetch and effective address calculation by the CPU, a floating-point instruction execution in the FPU can be divided into the following three phases:

- the BCU fetches the command and operand, the BCU fetches the FIA (floating-point instruction address);
- the FCU receives the operand from the BCU and converts IEEE format data into internal data format; and
- the ECU performs floating-point arithmetic.

By overlapping the operations for different floating-point instructions, each unit operates in parallel. The pipeline arbitration occurs in a finite-state machine that samples the state of each unit. The status of each instruction at any one time (see Figure 10) in the pipeline is as follows:

- the first instruction is in the ECU processing phase;
- the second instruction is in the FCU processing phase; and
- the third instruction is in the BCU processing phase.

Elementary functions. Designers know that their goals of high performance and high precision are often at odds with each other when designing floating-point units. Repeating iterations of converging algorithms ensures higher precision, yet the use of the same computational algorithm takes longer to execute. In pursuit of our design objective of higher precision in elementary functions, we attempted to streamline the Cordic algorithm to avoid lowering performance for precision. In realizing the elementary functions, we

1) adopted the Cordic algorithm because it lets us realize the entire range of elementary functions with a simple set of hardware.

2) added the adjustment mechanism to, for each exponent of the argument, always select the best set of angle constants used for rotations from a constant pool.

3) corrected errors after applying the Cordic algorithm to obtain higher precision with decreased Cordic iteration count.

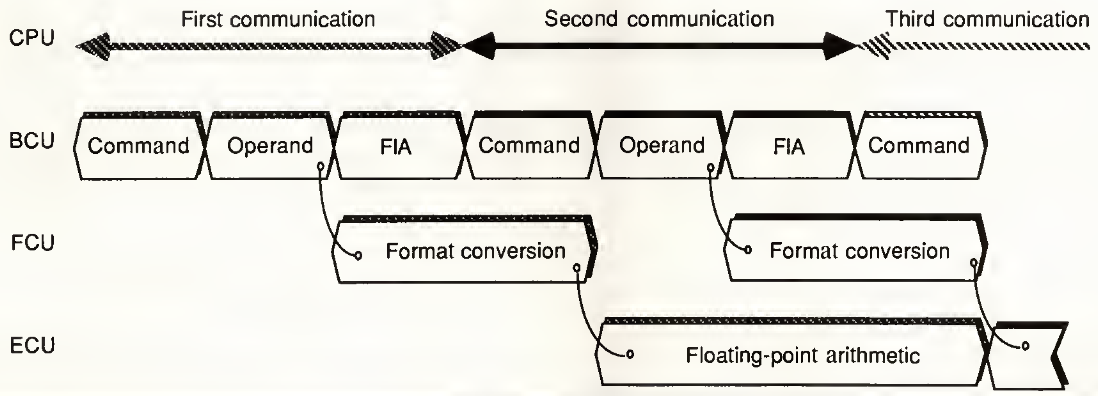
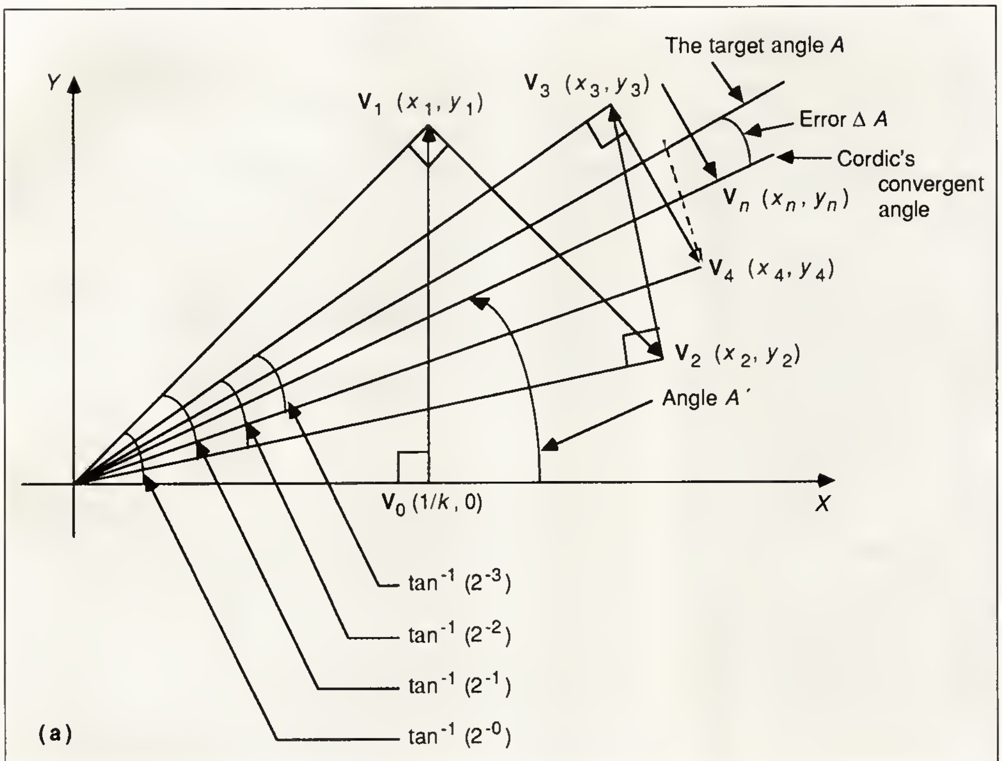


Figure 10. Pipeline timing in the Gmicro/FPU showing the CPU as it executes three consecutive memory-to-register floating-point arithmetic instructions. The length of each stage differs from one another. The format-conversion stage can start from the middle of the protocol. A state machine, which schedules each pipe to near-maximum efficiency, manages the pipeline.



$$\cos A = \cos(\Delta A) \bullet \cos(A') - \sin(\Delta A) \bullet \sin(A')$$

$$\approx \cos(A') - \Delta A \bullet \sin(A')$$

(b) for small ΔA

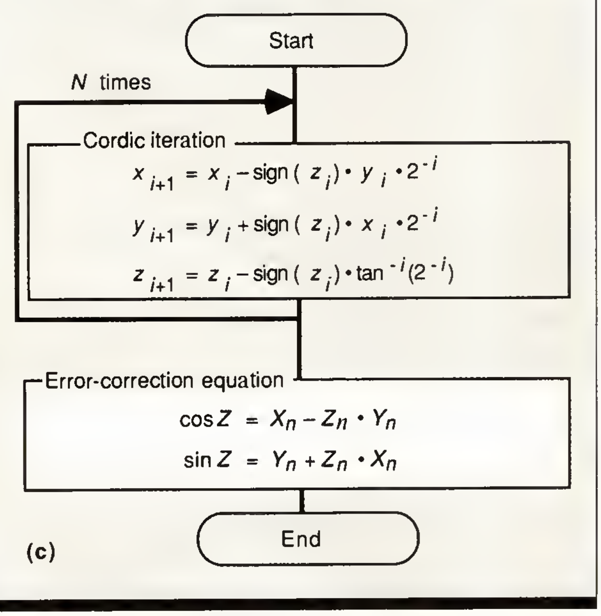


Figure 11. Applying the Cordic algorithm: the coordinate-rotation method (a); the error-correction method for a trigonometric function (b); and convergence improvement after replacing the last half of the Cordic iterations with a faster method (c).

In the following paragraphs we use the cosine function as an example to demonstrate how we implemented the Cordic algorithm.

Original Cordic algorithm. The Cordic algorithm invented by Chen[8] was expanded by Volder[9] and Walther[10] to be applied for the entire range of elementary functions. The algorithm's very wide range of functions even permits a multiplication to be performed with Cordic hardware. In the Gmicro/FPU, however, we use a separate multiplier for multiplication. The Cordic method obtains the solution by repeating vector rotations in the complex plane and eventually lets the vector converge to the angle corresponding to the argument for which the function value is searched.

Figure 11a illustrates the Cordic algorithm process of simultaneously obtaining cosine and sine for the angle A. Starting from the vector $\mathbf{V}_0(1/k, 0)$, we create another vector $\mathbf{V}_1(x_1, y_1)$ by rotation; we then form a perpendicular triangle with one vertex with the angle $\tan^{-1}(2^{-0})$ and one side as $\mathbf{V}_0$. Similarly, we obtain $\mathbf{V}_2(x_2, y_2)$ from $\mathbf{V}_1$, and $\mathbf{V}_3(x_3, y_3)$ from $\mathbf{V}_2$, and so forth to make the vector $\mathbf{V}_i$ approach the target angle A. We chose the direction of rotations in each iteration to make the vector closer to the target angle A. We obtain the coordinate of the vector after this rotation by the following iterative equations:

$$x_{i+1} = x_i - \text{sign}(z_i) * y_i * 2^{-i} \quad (1)$$

$$y_{i+1} = y_i + \text{sign}(z_i) x_i * 2^{-i} \quad (2)$$

$$z_{i+1} = z_i - \text{sign}(z_i) * y_i * \tan^{-1}(2^{-i}) \quad (3)$$

where we have chosen the plus and minus operators to converge to the target angle A. ΔA, the difference between the target angle and the vector $\mathbf{V}_n$, is translated into the error in the final solution. The maximum value of ΔA, $\max(|\Delta A|)$, becomes smaller by one bit with each rotation and eventually becomes less than 2^{-n} after n iterations:

$$\max(|\Delta A|) \cong 2^{-n} \quad (4)$$

To converge the rotational vector $\mathbf{V}_n$ to the target angle A with 64-bit precision, the same as that of extended double-precision format, the algorithm requires 64 iterations.

Cordic becomes faster. As the above argument shows, in the Cordic algorithm, we obtain only one significant bit of rotational angle per rotation. To obtain the solution more quickly, we leave the Cordic iterations after a certain number of rotations and obtain the cosine for the angle of a rotational vector $\mathbf{V}_n$, A'. This vector occurs in the vicinity of the target angle A with the difference ΔA. ΔA being sufficiently small, the following identity holds:

$$\begin{aligned}\cos(A) &= \cos(\Delta A) * \cos(A') - \sin(\Delta A) * \sin(A') \\ &\cong \cos(A') - \Delta A * \sin(A') \quad (5) \\ &\text{for small } \Delta A\end{aligned}$$

Using the identity in Equation 5, we obtain $\cos(A)$. The Cordic algorithm gives $\cos(A')$ and $\sin(A')$. We get ΔA from simple subtraction. Thus we can effectively replace the remainder of the Cordic iterations by a subtraction and a multiplication, which take very little time to execute with the Gmicro/FPU's multiplier.

Implementation, performance, precision. The Gmicro/FPU repeats the Cordic iterations 32 times and performs the error-correction operations seen in Figure 11b to obtain the cosine function. For all other functions including sine, tangent, and hyperbolic functions, similar error-correction equations exist, and with these the Cordic iterations repeat only 32 times.

Figure 11c summarizes the comparison of the vital statistics of the original Cordic algorithm and the improved Cordic algorithm by examining the cosine function with extended double-precision (80-bit) format. The Gmicro/FPU can execute one Cordic iteration in three machine cycles. By reducing the Cordic iteration count from its typical 64 times to 32 times, we save 96 machine cycles. The error-correction operation consisting of subtraction and 64 × 64 multiplication (which takes one machine cycle and 12 cycles respectively) can be overlapped with other operations. Thus the total microcode of the new algorithm takes 111 machine cycles as opposed to the conventional Cordic algorithm that takes 198 machine cycles. We achieve a performance improvement of 40 percent. See Table 3.

The preliminary evaluation of the precision tells us that the cosine function has, in most of its domain, 58 to 62 significant digits out of the 64 mantissa digits of extended double-precision data. Most libraries or floating-point processors have 53 to 58 significant digits. The difference is accounted for as follows:

- the Gmicro/FPU has 66-bit-long angular constants for Cordic operation, whereas most software libraries typically have 64-digit angular constants; and

Table 3. Cordic algorithm speed improvement on the cosine function.

Items	Conventional Cordic	Gmicro/FPU Cordic
Cordic iteration count	64	32
No. of multiplications	0	1
No. of significant digits	53-58 (out of 64)	58-62 (out of 64)
Latency times (machine cycles)	198	111

- the rounding-off error tends to accumulate with repetitive additions toward the end of the Cordic iterations. Thus, from a precision standpoint, the new algorithm demonstrates a satisfactory result.

Square roots. Square-root derivation is another area in which we can exploit the existence of the multiplier. With the Newton-Raphson algorithm we reach square roots extremely fast when fast multiplication is available. However, the IEEE floating-point standard sets a stringent requirement on the precision of the division and square-root solutions, which the N-R algorithm does not fulfill. A so-called precise solution is required for these operations. The IEEE standard states that the add, subtract, multiply, divide, square-root, and remainder operations "shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result . . . ."[11]

It has been believed that the above requirement essentially determines the computation algorithm for square-root and division operations. Most manufacturers today use the pencil-and-paper algorithm for these operations, which essentially obtains one significant bit of solution per iteration and which provides a set of intermediate results from which an ALU with infinite precision is effectively simulated.

Instead, the Gmicro/FPU uses the new algorithm to provide an IEEE square root with a smaller number of iterations. The FPU's N-R algorithm obtains a square-root solution in a certain neighborhood of the IEEE square root, which we call a *pseudo square root.* (See the following explanation.) It then reconstructs the intermediate result of the pencil-and-paper algorithm from the pseudo square root and begins the pencil-and-paper algorithm for the last several bits to obtain the IEEE square root.

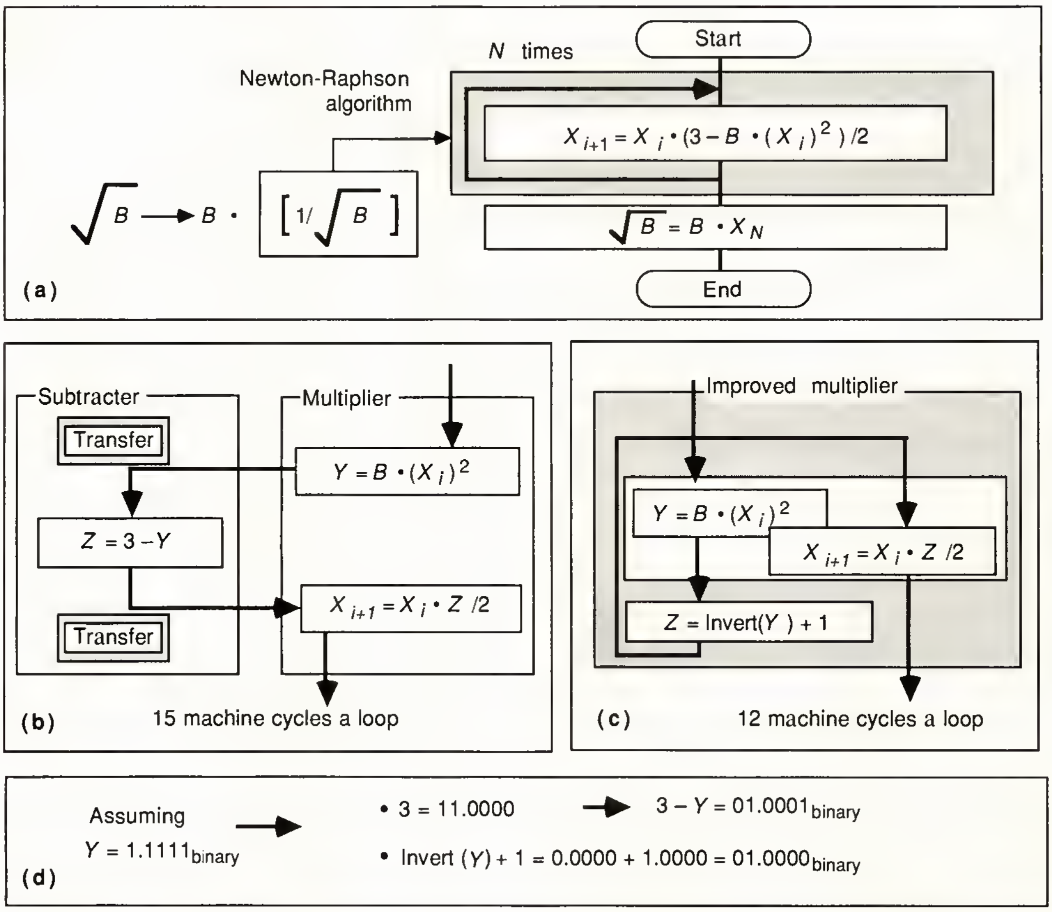


Figure 12. Square-root calculation. This technique for simplifying the Newton-Raphson method (a) saves three machine cycles in each iteration. The faithful evaluation in (b) is reduced further in (c). An example of 3 – *Y* = Invert(*Y*) + 1 (d).

Figure 12a shows the method of square-root extraction using the N-R principle. The N-R algorithm can obtain the reciprocal of the square root from an appropriate value by repeating Equation 6.

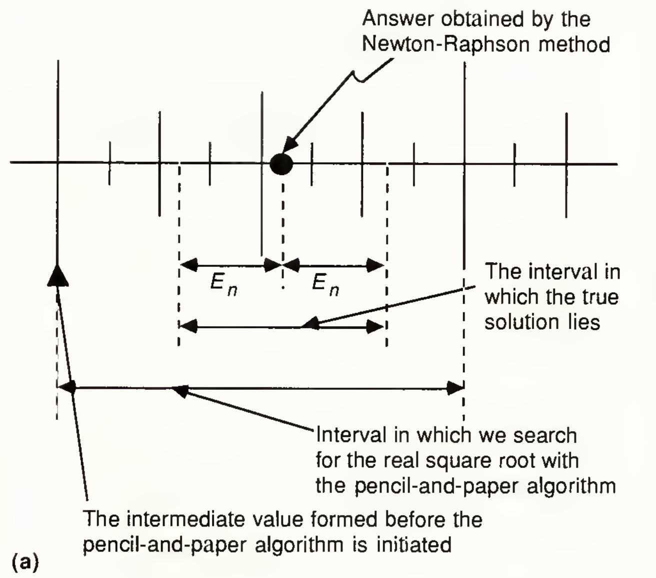
$$X_{i+1} = X_i * [3 - B * (X_i)^2]/2 \qquad (6)$$

where B is the source operand for the square root. To make the convergence faster, we look up in the ROM table an approximate value with 8-bit precision. With only three N-R iterations shown in Equation 6, we reach a result with 58-bit precision.

Shortening the estimating time. To reduce the amount of machine cycles per N-R iteration, we use the following techniques. Figure 12b shows that in an N-R iteration two data transfers between multiplier and subtracter are necessary, and that they cause the overhead of almost 30 percent of the machine cycles. We slightly tampered with the algorithm and replaced the subtraction operation with a complementing operation, which we show in Figure 12c. This operation can be realized with a small amount of hardware. The expression $3 - B *(X_i)^2$ is very close to a [one's complement of $B * (X_i)^2$] + 1.0. An error with the weight of one LSB (least significant bit) creeps in, yet it can be proven that its effect is essentially negligible. The subtraction $3 - B\ \Sigma\ (X_i)^2$ can be achieved only by inverters installed in the multiplier. This improvement reduces the overhead by 30 percent and the N-R iteration completes in 12 cycles.

Based on a pseudo solution obtained by the N-R method, the LSBs of the mantissa are recovered, and the classical restoring method of the square-root algorithm starts near the LSB. When the iterations of the restoring method complete, intermediate values are collected and a sticky bit is determined. The rounding operation yields the result required by the IEEE floating-point standard.

Figure 13a illustrates how the pencil-and-paper algorithm is actually restarted. The pseudo square root obtained by the N-R method is in the neighborhood of [pRoot − E_n, pRoot + E_n], where the real square root



(a)

```
function pencilAndPaperSquareRoot(rX);

}
This is a classic method to obtain a square root used in most IEEE conforming
floating-point arithmetics. The parameter rX lies in the interval (1.0, 2.0)
to simplify the argument. The indices are asignad for mantissa bits in the
following manner :

  0   -1  -2                   -(q-1)                -N+1  N  -N-1
+---+---+---+------------------+---+--------+---+---+---+
| 1 )                                               | R | S |
+---+---+---+------------------+---+--------+---+---+---+
    ^ binary point
}

     function searchRoot(iRoot,iResidue, k);
{
searchRoot add valid -k bit to square root solution and return result. N+1
Recursive invocations yield N+2 bits of solution.
}
        begin
        if (k = N + 1)
             then if (iResidue = 0) searchRoot := iRoot
                  else begin
                       searchRoot := iRoot + 2^-N-1;
                       iResidue := newResidue(iRoot, iResidue);
                       end
        else if (root is in lower half of the interval)
             then searchRoot := searchRoot(iRoot, iResidue, k+1)
             else searchRoot := searchRoot(iRoot + 2^-(k-1), iResidue, k+1);
        end;

    begin
    iRoot := 1.0;
    rRoot := searchRoot(iRoot, 0.0, 1);
    pencilAndPeperSquareRoot := round(rRoot);
    end;
```

(b)

```
function hybridSquareRoot(rX);

{    pRoot : pseudo root obtained by Newton-Raphson algorithm
     iRoot1 : a candidate intermediate value to restart pencil/paper method
     iRoot2 : a candidate intermediate value to restart pencil/paper method
     iX1 : square value of iRoot1
     iX2 : square value of iRoot2
     rRoot : root solution with ground and sticky bits
}

     function searchRoot(iRoot, iResidue, k);
{         determines one more valid bit -k of root and return result
}

     function newtonRaphson(rX);
{         returne pseudo solution pRoot lying in neighbourhood
          (rRoot-En, rRoot+En) of real solution rRoot (|pRoot - rRoot| ≤ En).
          An integer quantity q used in other part of the program is an integar
          such that e < 2^-q+1.
}
     function clearBits(intValue, l, m);
{         clear bits l through m of intValue
}

     begin
     pRoot := newtonRaphson(rX);
     iRoot1 := clearBits(pRoot, -p, -n-1);
     iX1 := iRoot1 * iRoot1;
     iResiduel := rX - iX1;
     if (iResiduel = 0) then rRoot := iRoot1
     else if (iResiduel > 0)
          then begin
               iRoot2 := iRoot1 + 2^-(q-1);
               iX2 := iRoot2 * iRoot2;
               iResidue2 := rX - iX1;
               if (iResidue2 = 0) then rRoot := iRoot2
               else if (iResidue2 > 0)
                    then rRoot := searchRoot(iRoot2,iResidue2, q)
               else rRoot := searchRoot(iRoot1, iResiduel, q);
               end
          else begin
               iRoot2 := iRoot1 - 2^-(q-1);
               iX2 := iRoot2 * iRoot2;
               iResidue2 := rX - iX1;
               if (residue2 = 0) then rRoot := iRoot2
               else if (residue2 < 0)
                    then rRoot := searchRoot(iRoot2, iResidue2, q)
               else rRoot := searchRoot(iRoot1, iResiduel, q);
               end;
    hybridSquareRoot := round(rRoot);
    end;
```

(c)

Figure 13. Hybrid algorithm to obtain an IEEE square root. Before starting the pencil-and-paper method (b), one must make the interval in which the real square root lies (a) the subset of the interval of convergence. The hybrid square-root algorithm used in the Gmicro/FPU (c). For a machine with a parallel multiplier, a square root can be obtained faster.

lies. The [pRoot $-$ E_n, pRoot + E_n] neighborhood is again a subset of some interval within which the IEEE square root can be determined by the pencil-and-paper method. Figure 13b shows the Pascal-like outline of the conventional pencil-and-paper algorithm to obtain the IEEE square root. Figure 13c shows how its shortcut reconstructs the intermediate values for the pencil-and-paper method from the pseudo root obtained by the N-R algorithm. The pencil-and-paper algorithm starts from near the LSB.

Exception handling. One of the areas in which programming techniques cannot conceal the shortcomings of hardware is that of exception traps. When the function of an exception trap is realized with software, the overhead is enormous and visible. The Gmicro/FPU provides trap facilities in all operating modes, and no information is lost in a floating-point arithmetic exception, relieving programmers from any apprehensions about exception handling. The FPU

- *contains a command pipeline that does not alter the command stream.* Even though the processing is done in parallel, the sequential execution model is retained. Programmers do not have to pay attention to pipelining unless an exception trap occurs.
- *retains the information such as the instruction codes, instruction address, all operands that were used in the instruction, and exception flags.* The user can extract the flags with the floating-point-state save instructions FSAVE and FSTC. From the point at which the exception occurred, the user can deduce the cause of the problem.
- *retains the state in all pipes of the pipelines when an exception occurs.* The FPU also retains all instructions in process. Using this information, programmers can fix the problem or even restart the entire program from the point of exception after fixing some of the intermediate values.

Fortran to generate faster code.

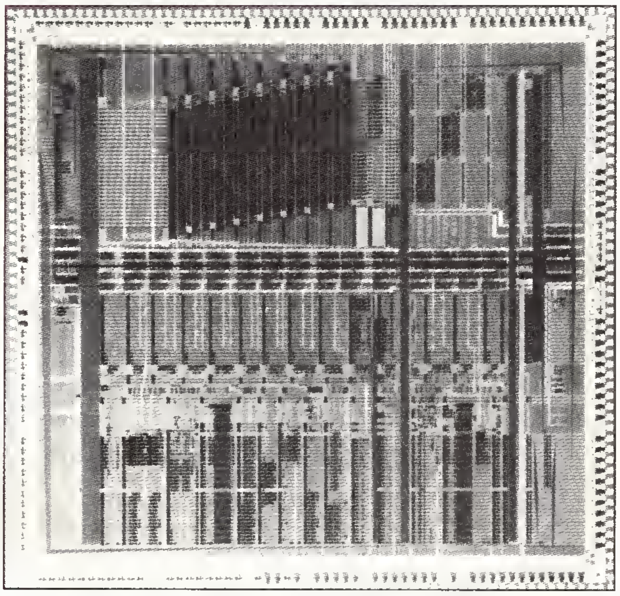
determining the system performance.

Each processor is formed as a pipelined configuration of the five-chip set. When process locality is assumed, slower off-chip performance is not crucial to overall system performance because inter-chip communication occurs infrequently.

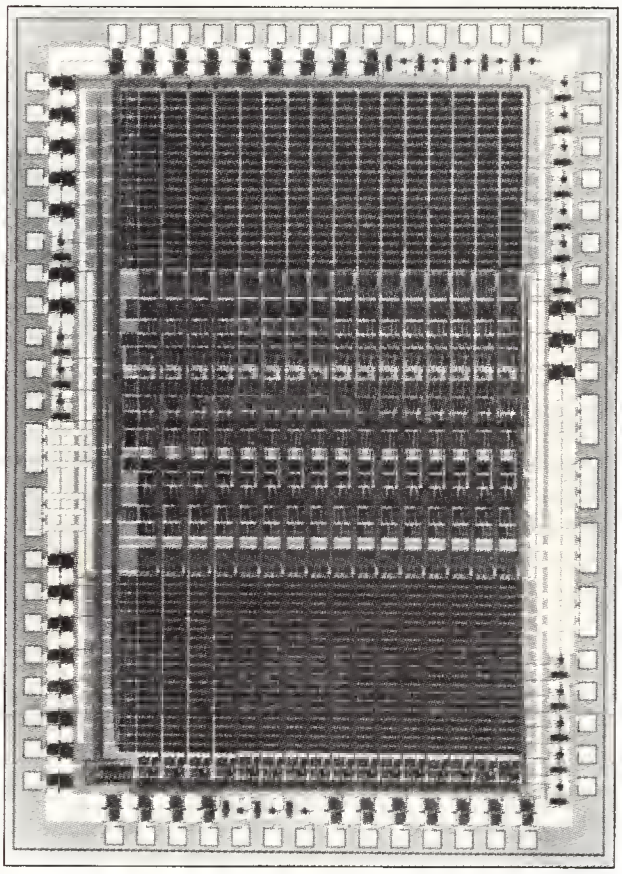
form a processing pipeline through which data travels in a packet format (see Figure 4a). Each chip can be used repeatedly to build a PE. The joint and branch unit also connects PEs in the system. The photo in Figure 4b shows a prototype of the single-board, data-driven

Data-driven processors As shown in Figure 3a, the Manchester dataflow

Data-driven microprocessor



(a)



(b)

Figure 5. Chip photomicrographs of the functional processor (a) and queue buffer (b).

computer. Figures 5a and 5b contain photomicrographs of the functional processor and queue buffer chips. Table 1 describes the chip set.

Figures 6a and 6b provide the formats for input and operation packets. Input packets come from the off-chip area and include the resultant packets generated from the functional processor. Operation packets are sent from firing control to the functional processor. Each packet is organized in a two-word format with a header word and a tail word. A selection code field in the header word contains such information as packet type (that is, whether it is an initializing packet for program and data loading or an execution packet) and physical destinations (on-chip or off-chip) of the packet. A color/generation identifier indicates the environment or context to which the packet belongs. The most significant bit of the identifier denotes whether it will be used as a color or generation identifier. A destination-node identifier serves as part of a keyword in the matching-memory access as well as an input address of the cache-program store. Depending on the retention of the color/generation identifier, multiple sets of the input data can share an identical function/program.

Figure 7 provides examples of the concept of multiple-generation data processing and concurrent processing of a common function. Packets belonging to other generations are processed concurrently, which could lead to the possibility of inconsistent results. However, the program executes consistently because the firing control fires only when two packets have the same generation identifier. Unlike the static architecture in which the data process executes only for a single color/generation identifier, this type of *dynamic* architecture allows multiple packets of different color/generation identifiers to occupy the same primitive node concurrently. By exploiting this architectural feature, each program/function's parallelism can be superimposed on the processor, leading to an efficient utilization of resources. Even if a program is not highly parallel, effective parallelism in the processor is the product of the number of parallelisms times the number of concurrently applied generations. This process results in efficient pipeline processing because the maximum performance is obtained when all pipeline stages are filled with concurrently activated data.

To improve the packet-flow rate through the pipeline, each function block further subdivides into several pipeline stages. An input packet enters the joint and branch chip and merges with the packet stream in the circular pipeline. Cache-program store updates the packet header after fetching the next operation code and destination address. In the firing-control chip, with the color/generation code and destination address as keywords, the packet waits for its corresponding partner. Consequently, the firing control can be viewed as an associative memory in which two packets that have an identical keyword associate with one another to generate an operation packet. The matched operation packet is then sent to the functional processor, in which the pipelined operation proceeds according to the op-

Table 1.
Description of the five-chip set.

Chip	Performance	Pin count	Transistor count	Function
Functional processor	40 Mflops	208	85,000	Arithmetic operation on 32-bit data (floating-point/integer)
Queue buffer	20 MOPS	208	20,000	Buffer memory
Joint and branch	20 MOPS	180	13,000	Packet-transmission control
Firing control	20 MOPS	208	450,000	Matching operation of operands
Cache program	20 MOPS	208	400,000	Program memory store

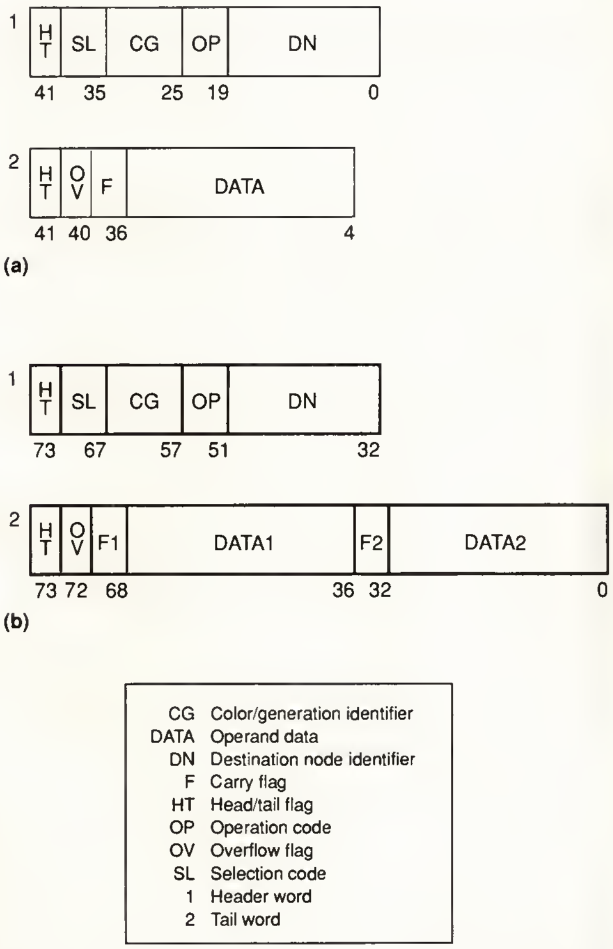


Figure 6. Organization of the input (a) and operation (b) packets.

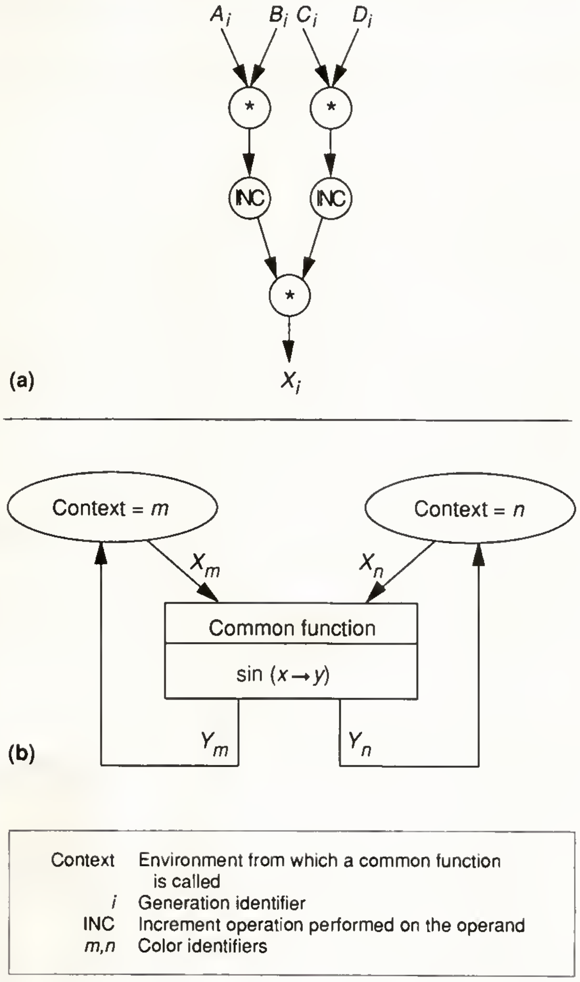


Figure 7. Illustration of the color/generation identifier's function in multiple-generation data processing (a) and concurrent processing of a common function (b).

eration code. The queue buffer absorbs any fluctuation in the packet stream. A packet entering the joint and branch chip after going through the queue buffer either continues to circulate within the pipeline or exits from it depending on the value of its selection code. Each lap around the circular pipeline corresponds to the execution of one operation.

This dataflow processor extensively utilizes *an elastic-pipeline* processing scheme, which we later discuss in detail. Its floating-point processing unit achieves a peak performance of over 40 million floating-point operations per second (Mflops).[9] We subdivided this function into 12 pipeline stages to aid performance.

Because data is interdependent in conventional sequential processors, an interlocking phase and pipeline flushing frequently occur between pipeline operations. This phase recovers the previous status when a Branch instruction is actually executed. Therefore, as the number of pipeline stages increases in conventional processors, the cost and complexity of pipeline control also increase rapidly. For this reason, subdividing the pipeline function beyond five to seven stages for sequential processors does not improve performance.

Data-driven processors, however, do not concurrently activate operations that are mutually dependent. (Mutual dependency in this case means the inability to generate one of two packets until the other packet has completed execution.) The firing principle permits unordered execution of the fired operation. This procedure allows execution to be carried out without any side effects once the operating packet is produced in the firing control.

VLSI hardware

A notable feature of our data-driven processor is that each type of chip in the five-chip set employs a unique processing structure: a pipeline with an elastic data-buffering capability.[10,11] Figure 8 shows the structure of the elastic-pipeline concept. The operation packet consists of operation code and operand data. An operation code is processed by the decoders placed at each stage of the pipeline and determines the function of the hardware primitives located between data latches. The latches store the resultant packet until the succeeding

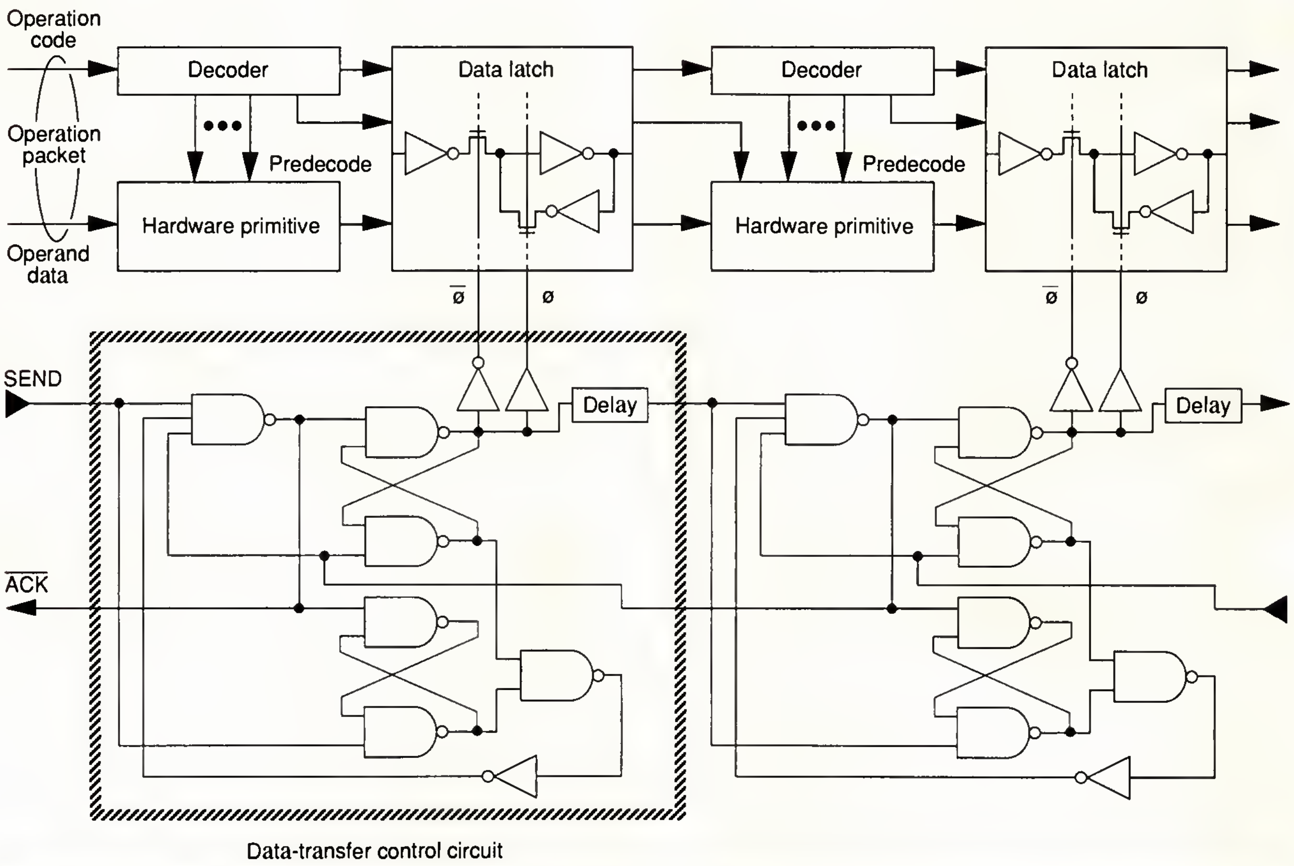


Figure 8. Structure of the elastic pipeline.

stage becomes vacant. The decoder predecodes the operation code at each pipeline stage to minimize delays. The number of stages depends on the complexity of the function. Data transfer between the latches uses the handshake mode of transfer by exchanging the SEND and Acknowledge signals between successive, self-timed, data-transfer control circuits. This method achieves a self-timed circuit free of any system clocks. Delay elements are purposely added to the SEND signal line to guarantee an appropriate data-processing time through the logic circuit between the latches.

The elastic pipeline provides the following characteristic features to a data-driven microprocessor.

Elasticity. Because of the elastic mode of data transfer through the pipeline, data latches alternatively hold data during a data transfer or empty it to flow continuously through the pipeline. On the other hand, if the data begins to congest the pipeline, the empty latches between the data are "squeezed together" when the empty pipeline stages fill with accumulating data. These latches then act as data buffers (see Figure 9). This buffering capability is also favorable for the data-driven processor because the production rate of the operation packets, which contain operand pairs, usually fluctuates around an average.

Noise reduction. Due to simultaneous switching of all the transistors in clock-synchronous systems, a peak current demand causes unfavorable noise along the power distribution lines both on and off chip. This peak demand may introduce disruption of the next clock cycle due to a logic hazard. This hazard is created by the excessive inductive noise during a sudden surge. The problem becomes serious as the clock frequency increases. In the elastic-pipeline scheme, however, transistor switching is determined by the amount of handshaking that occurs distributively along the elastic pipeline. Obviously, this procedure smooths out the peak current demand and significantly decreases inductive power-line noise on a VLSI chip.

Clock-skew-free design. Along with the inductive power-line noise problem, clock skew also becomes a crucial design problem at increased clock frequencies. Some conventional processors centrally control communication between each function block through a long, common, passive-metallic bus. In this case, the skew between the clock and control signals can grow significantly. This skew substantially limits the number of logic stages that can safely be placed between the latches. In the elastic-pipeline processor scheme, function blocks are distributively controlled by neighboring function blocks. Asynchronous communication between successive stages is completely localized. Consequently, the scheme is entirely free from the global skew problem encountered in conventional processors. In terms of logic-design tasks, this means that the designer is freed from clock-distribution problems.

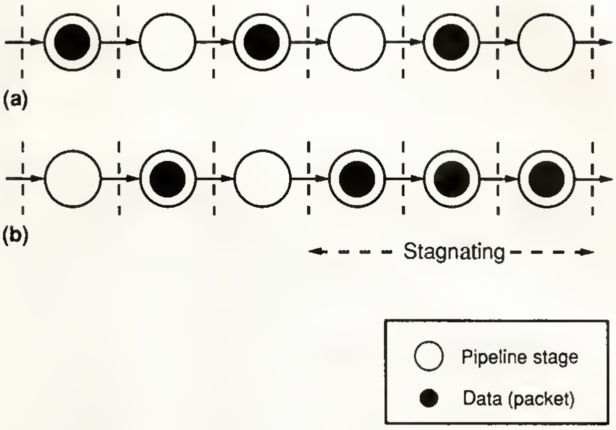


Figure 9. Elastic data in flowing (a) and congested (b) states.

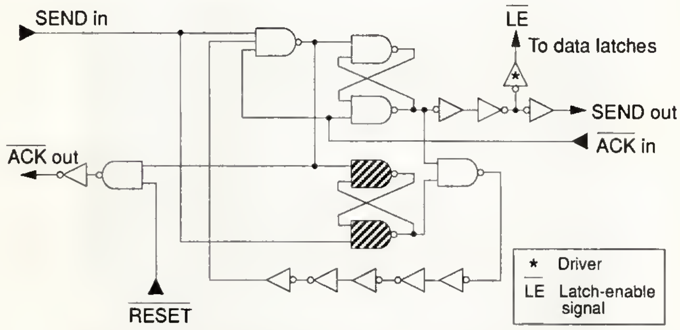


Figure 10. Self-timed, data-transfer control circuit.

Figure 10 shows an example of a self-timed, data-transfer control circuit, which is a key component in the elastic pipeline.[12]

Figure 11 on the next page shows a test chip used for verification and evaluation of the control circuit. The test chip has a data throughput rate of well over 50,000,000 words per second. Figure 12 demonstrates that it takes 19.52 nanoseconds to transfer one word, which corresponds to one cycle of the latch-enable SEND signal. One of the data bits output from the latch shows the stabilized transfer. Since this chip is clock free, the data-bit signals swing according to the SEND signal. In conventional clock-synchronous circuits, the SEND signal would correspond to the clock.

This high data rate occurred even though the chip was fabricated by a rather conservative, 1.3-micrometer, complementary metal-oxide semiconductor (CMOS) process.

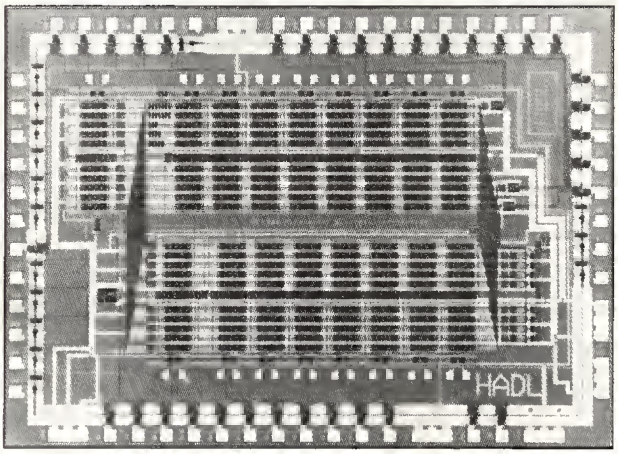


Figure 11. Test chip with an elastic-pipeline structure.

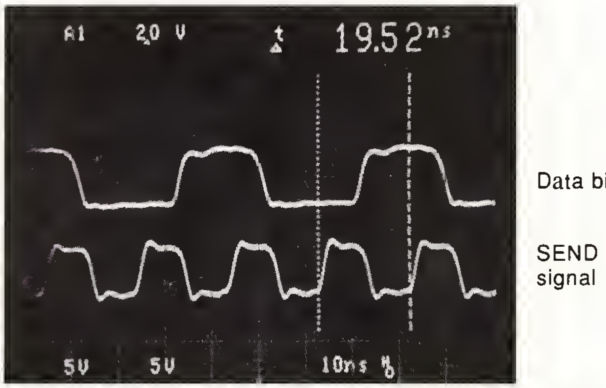


Figure 12. Data-throughput measurement in a test chip.

The data-transfer rate for the self-timed control circuit is shown by

$$\text{Rate} = 1/(t_F + t_B)$$

where t_F is a forward-propagation delay of the SEND signal and t_B is the backward-propagation delay. To improve the transfer rate while keeping the processing delay time constant at each stage, it is necessary to minimize t_B. We propose an improved transfer-control circuit as shown in Figure 13 to minimize the t_B.[10]

The elastic-pipeline scheme applies not only to the ALU but also to various functional blocks in the data-driven processor. One example is the pipelined firing-control configuration shown in Figure 14. For each incoming result packet the lower four bits in the color/generation identifier and the lower six bits of the destination field are concatenated to yield a hashing address to the firing control. A presence bit of the read-out data in matching memory indicates whether any result packet has already been written in the address. If the result packet's corresponding partner is not found, the input result packet is stored in the same address as the read-out hashing address and turns on the presence bit. When the partner is found, an operation packet containing a pair of operands is generated by merging the two result packets.

In the firing control, we employed the pipelined-processing scheme at two different levels. At the macro level, the firing control subdivides into three units: prematching, matching memory, and packet assembly. In the prematching unit, the comparator processes tag information (that includes color/generation and destination-node identifiers) in two simultaneously input packets. This process avoids duplicated, concurrent write access onto the same address from two input ports. The matching-memory unit has a two-input/two-output multiport memory and can execute two queries simultaneously. In the packet-assembly unit, an asyn-

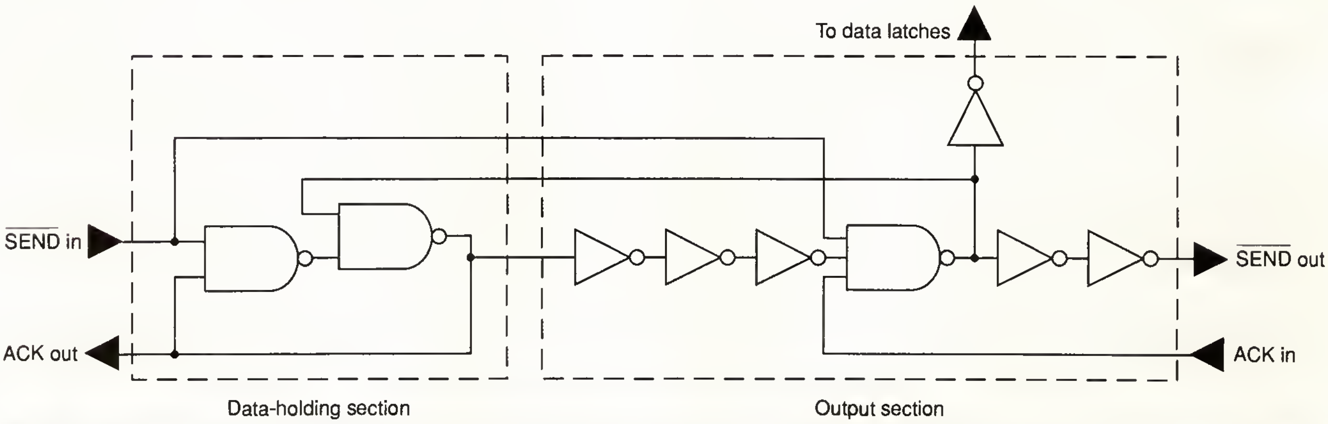


Figure 13. Advanced self-timed, data-transfer control circuit.

chronous arbitration circuit merges the upper and lower packet streams into one output packet stream.

Each macro-level section further subdivides into several pipeline stages. For example, the matching-memory section has four elastic pipeline stages. In the first stage, two prematched packets from the upper and lower paths merge into just one path. Memory reads occur in the second stage, while the third stage compares tag information between the read-out data and the input packet. A memory write can take place in the last stage according to the result of the comparison.

Language processing

As previously mentioned, the data-driven processor executes a class of dataflow instructions based on the diagrammatical data-driven language, D3L. Figure 15 shows the flow of the language-processing system in which a load module in machine code is generated as a combination of C-like textual source programs and assembly-language representations.

Compiler. We developed a compiler that

- checks the syntax of the text and extracts control structures,
- analyzes data dependencies among variables in the program and produces a data-dependency list, and
- generates an object module based on the data-dependency list.

The compiler automatically appends the operations needed for the control structure such as loops and conditional processes.

Assembler. In addition, an object module can be directly generated from textual assembler source programs in which data dependency is directly specified in a textual format. This approach lets the programmer optimize program structures. Current compiler development does not ensure that the compiler will generate the best structure to critical portions of the program.

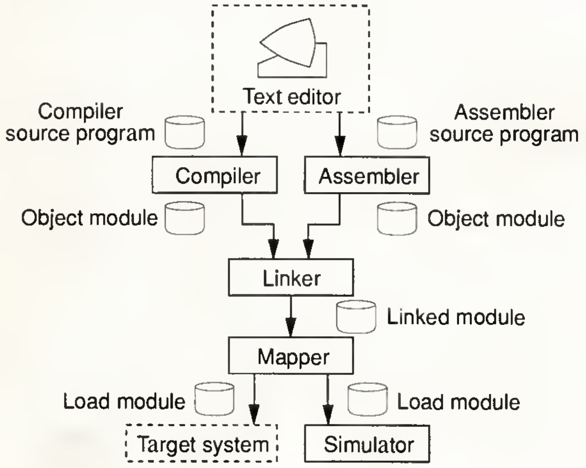


Figure 15. Flow graph of C-like and assembly-language-processing system.

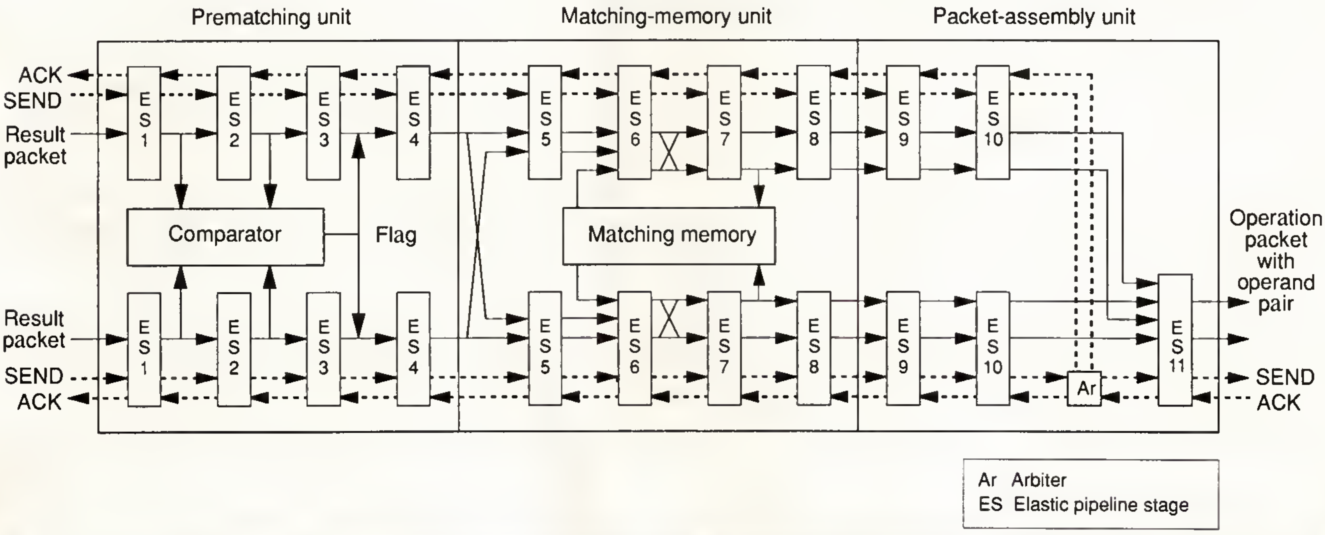


Figure 14. Block structure of the firing-control unit.

```
main(ix, ia1, ia2, ia3, ia4;oy, oz)
/*------input------*/
input     int     ix, ia1, ia2, ia3, ia4;
/*------output-----*/
output    int     oy, oz;
/*-----process-----*/
{
          int     x3, x2, x1;

          x1=ix;
          x2=x1*ix;
          x3=x2*ix;
          oy=ia1*x3+ia2*x2+ia3*x1+ia4;
          oz=ia1*x3-ia2*x2-ia3*x1-ia4;

          return(oy, oz);
}
```

(a)

```
          func      main
          %-------input
          x:        input       in1
          a1:       input       in2
          a2:       input       in3
          a3:       input       in4
          a4:       input       in5
          %-------output
          y:        output
          z:        output
          %-------process
          in1:      nop         f10, f11
          in2:      nop         f21*
          in3:      nop         f11*
          in4:      nop         f22*
          in5:      nop         f41*
          f10:      nop         f20, f21
          f11:      mul         f22
          f20:      nop         f30, f31
          f21:      mul         f40*
          f22:      add         f31*
          f30:      nop         f40, f50
          f31:      mul         f41
          f40:      mul         f50*
          f41:      add         f60*, f61*
          f50:      mul         f60, f61
          f60:      add         y
          f61:      sub         z
```

(b)

Figure 16. Programming-language examples: C-like (a) and assembly (b).

Linker. Because object modules generated by the compiler and the assembler naturally have the same format, these modules can be freely linked into a load module. A rank analysis also occurs to determine critical path lengths from the input nodes to each of the primitive nodes within the module. The result passes to a mapping program, which approximates the order of execution.

Mapper. A user can specify to the mapper the functional-level mapping of load modules onto different individual processes in a multiprocessor system. The mapper can also automatically map the nodes to individual processors using rank-analysis results.

Simulator. We developed a simulator to evaluate the effect of load-module mapping onto processors. The simulator attunes fine-grain mapping of a target module with various topologies in multiprocessor systems.

Textual source programs. Figures 16a and 16b show examples of source programs written in C-like and assembly languages. These two programs have exactly the same meaning. Dataflow graphs that are equivalent to the load module generated from C-like or assembly-language programs are shown in Figure 17a and 17b. Rank refers to the critical path length of the node, counting from the input node. The path length is determined by the number of arcs between these two nodes. The load module generated from the assembly-language program shows better runtime efficiency.

Applications

Figure 18 shows the processor's support environment for developing application programs. This environment is constructed by the host computer and the emulator, which connect to either an RS-232C interface or a general-purpose interface bus (GPIB). The emulator consists of the data-driven processor on the board computer and such peripheral boards as external program store, external data memory, and host interface. Table 2 on the next page summarizes board specifications. The host computer achieves program loading and collection of the result data.

A number of potential applications for the data-driven processor exist.

Real-time processing of high-flow-rate data streams. Because of the dynamic data-driven scheme employed in the processor, one of the most promising fields for applications is real-time signal processing in which input data reaches the processor in data-stream form. High-speed signal and graphics processing are two potential applications for the processor.

Multilevel event-driven processing. Since the data-driven processor is completely free from context-switching overhead, the processor exhibits very fast responses to external world events. This feature promises high performance in applications such as robotics and machine control in which multilevel event-driven capability is crucial.

Computationally intensive problems. The feasibility of using a packet-oriented, data-driven processor in multiprocessor organization suggests that many computationally intensive programs can be mapped on a data-driven multiprocessor system. Applications include the solution of partial differential equations and the emulation of neural networks.

One of the major claims of the data-driven processor is that it is a scheme inherently free from any side effects. Another claim is that the processor's visual representation of programs enhances programmer productivity. To achieve these claims, one absolutely needs a comfortable programming environment in which the graphical specification of a target problem is easy to perform. Therefore, high-level graphical specifications are necessary to popularize the data-driven programming paradigm. We continue to investigate these needs.

Although we observed a peak performance of 40 Mflops in the elastic pipeline and designed the one-board, data-driven computer to operate at a speed of 20 Mflops, the entire system operates at 13 Mflops. The performance degradation of the entire system from on-chip processes apparently reveals a heavy communication penalty due to off-chip data transfers. The highest flow of data through the processor is exposed to speed-constrained buffer drivers. Clearly we have to integrate essential functional blocks of the processor onto a

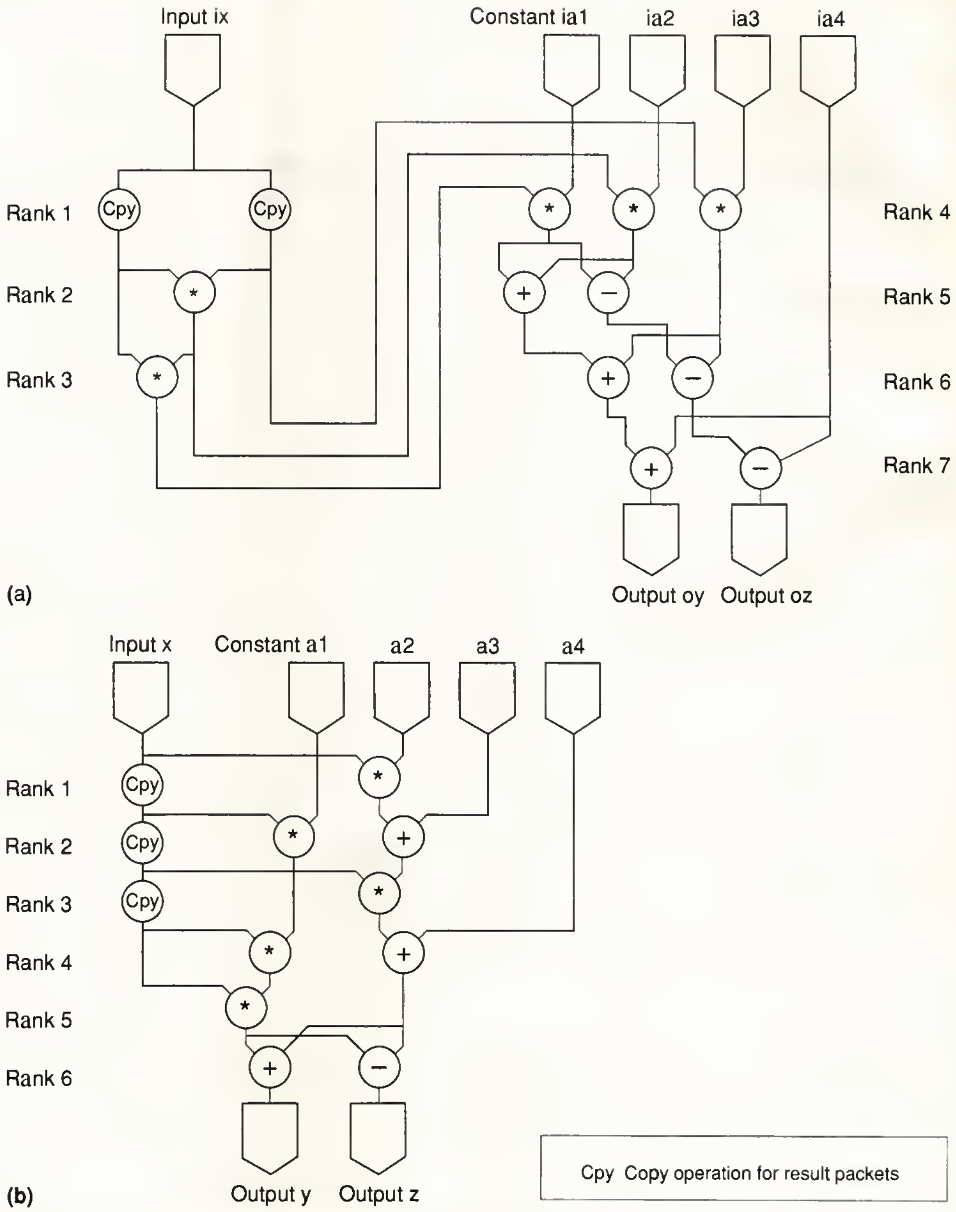


Figure 17. Dataflow graph for a C-like (a) and assembly-language (b) programs.

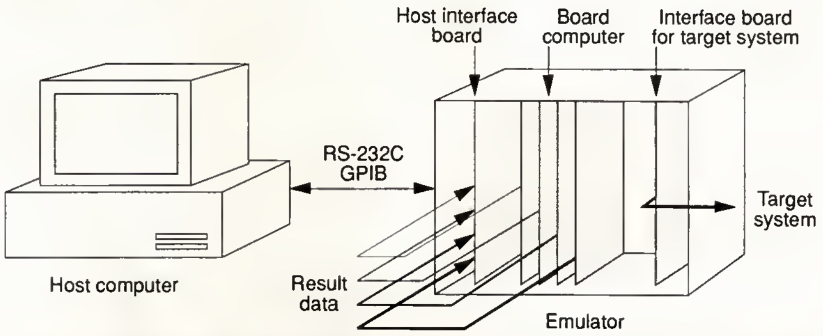


Figure 18. Development-support environment.

Type	Data
Performance	20 Mflops
Data length	32 bits
Data type	Floating-point/integer
Program memory size	Internal: 1-Kbyte steps External: 512-Kbyte steps
Data memory size	External: 4 Gbytes
External memory access rate	10 Mwords/second (1 word = 4 bytes)
I/O interface	
Multi PE connection	Two 42-bit in ports Two 42-bit out ports
External program ports	One 42-bit data port One 74-bit address port
External data ports	One 42-bit data port One 58-bit address port
Board size	48 × 38.5 centimeters

Table 2.
Board specifications.

single VLSI chip, and we are attempting to do that. We are also working on improving the instruction-set architecture of the processor.

Acknowledgments

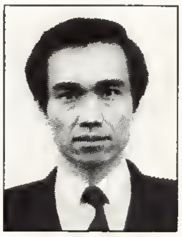
Although it is impossible to thank individually all those who assisted with this research and development project, we express our sincere appreciation to all our colleagues. Without their endeavors, this processor would not exist. We also gratefully acknowledge T. Shichiku for his efficient support in proofreading our manuscript.

References

1. J.A. Sharp, *Data-flow Computing*, Ellis Horwood Ltd., Chichester, England, 1985.
2. H. Nishikawa et al., "Architecture of a One-Chip Data-Driven Processor: Q-p," *Proc. 16th Int'l Conf. Parallel Processing*, The Pennsylvania State University Press, University Park, Penn., Order No. FJ783, IEEE CS Press, Los Alamitos, Calif., Aug. 1987, pp. 319-326.
3. D. Pountain, "Microprocessor Design—The Transputer and Its Special Language, Occam," *Byte*, Vol. 9, No. 8, Aug. 1984, pp. 361-366.
4. J. Backus, "Can Programming Be Liberated from the von Neumann Style?" 1977 ACM Turing Award Lecture, *Comm. ACM*, Vol. 21, No. 8, Aug. 1978, pp. 613-641.
5. W. Myers, "DeMarco Foresees Change to Parallel Processing," *IEEE Software*, Vol. 5, No. 2, Mar. 1988, pp. 92-93.
6. J.R. Gurd, C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 34-52.
7. J.B. Dennis, G. Gao, and K. W. Todd, "Modeling the Weather with a Data Flow Supercomputer," *IEEE Trans. Computers*, Vol. C-33, No. 7, July 1984, pp. 592-603.
8. T. Yamasaki et al., "VLSI Implementation of the Variable Length Pipeline Scheme for Data-Driven Processors," *Digest Symp. VLSI Circuits*, Aug. 1988, pp. 31-32.
9. S. Komori et al., "A 40-Mflops 32-Bit Floating-Point Processor," *Digest Int'l Solid-State Circuits Conf.*, Feb. 1989, pp. 46-47.
10. H. Terada et al., "VLSI Design of a One-Chip Data-Driven Processor: Q-v1," *Proc. Fall Joint Computer Conf.*, IEEE CS Press, Los Alamitos, Calif., Oct. 1987, pp. 594-601.
11. K. Asada et al., "Hardware Structure of a One-Chip Data-Driven Processor: Q-p," *Proc. 16th Int'l Conf. Parallel Processing*, Aug. 1987, pp. 327-329.
12. S. Komori et al., "An Elastic Pipeline Mechanism by Self-Timed Circuits," *J. Solid-State Circuits*, Vol. 23, No. 1, Feb. 1988, pp. 111-117.

Shinji Komori is a senior engineer in the LSI Research and Development Laboratory at Mitsubishi Electric Corporation. He is currently researching VLSI-oriented, data-driven architectures and developing data-driven microprocessors.

Komori received the BS degree in applied mathematics and physics from Kyoto University. He is a member of the Institute of Electronics, Information and Communication Engineers of Japan (IEICE).



Kenji Shima is a senior engineer in the Industrial Systems and Electronics Research and Development Laboratory at Mitsubishi Electric Corporation. He has been involved in the research and development of sound synthesis, voice analysis, and voice recognition since 1974 and is now interested in developing applications for data-driven processors.

Shima received the BS and MS degrees in electrical engineering from Kobe University. He is a member of the IEICE and the Acoustical Society of Japan.

Souichi Miyata is manager of the data-driven microprocessor development group in the Integrated Circuits Group at Sharp Corporation. He has experience in the development of memory and logic devices and has been researching and developing highly parallel processing systems since 1982.

Miyata holds a Doctor of Engineering degree in material sciences from Osaka University. He is a member of the IPSJ and Japan's Society of Applied Physics.

Toshiya Okamoto is a senior engineer in the Integrated Circuits Group at Sharp Corporation. He has been researching and developing highly parallel processing systems since 1982. He currently develops dataflow microprocessor architectures and software tools.

Okamoto received the BS and MS degrees in system engineering from Kobe University. He is a member of the IPSJ.

Hiroaki Terada is a professor at Osaka University where he is responsible for research and education in digital systems and circuits. His current research activities include developing a diagrammatical data-driven language, a data-driven architecture, and its VLSI-oriented implementation. He was a visiting professor at the University of Essex, England, and a visiting scholar at the Centre National d'Etudes des Telecommunications, Lannion, France.

Terada received the BE degree in electrical engineering from Ehime University and the ME and PhD degrees in electronic communications from Osaka University. He has served as chair of the Technical Group on Switching Engineering, IEICE, and as a member of the Board of Directors of the IPSJ. He received the IEICE Achievement Award in 1988 and the IEICE Kobayashi Memorial Achievement Award in 1989.

Questions concerning this article can be directed to Shinji Komori, Advanced Microprocessor Development Department, LSI Research and Development Laboratory, Mitsubishi Electric Corporation, 4-1 Mizuhara, Itami, Hyogo Japan 664.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 156 **Medium** 157 **High** 158

SPECIAL FEATURE

The Transputer T414 Instruction Set

Have you wondered how to decipher the transputer instruction set? Here's your chance to compare it with something you know.

Jean-Daniel Nicoud
Andrew Martin Tyrrell

Ecole Polytechnique Federale de Lausanne

Multiprocessor systems offer a number of advantages over the single-processor configuration. However, multiprocessor systems require interprocessor communication, some form of time-slicing, and primitives for parallel operation. The transputer is a microprocessor designed specifically for a multiprocessor environment. Because of this, many of the instructions performed by the transputer are unfamiliar to designers of single-processor systems. In addition, Inmos Ltd.—the manufacturer of transputers—uses very short notations with implied operands within its assembly language. This approach differs greatly from the techniques used for 32-bit microprocessors like the ones in Motorola's M68000 and National Semiconductor's NS32000 families.

Consequently, we wrote this article as a tutorial on the instruction set of the T414 transputer and include assembly-language notations that are more familiar to microprocessor-system designers than the Inmos documentation.

Transputer types

All types of transputers have the same basic instruction set. The IMS T414 and T800 transputers are powerful, pin-compatible, 32-bit processors especially designed for implementing parallel architectures. *IEEE Micro* has already covered the hardware and general-software aspects of these transputers.[1] Note that both the T414 and the T800 have four, high-speed serial links supported by internal direct-memory access (DMA) channels, two timers to assist in switching tasks, and internal static RAM (Figure 1). The T800 differs from the T414 by having 4 Kbytes of RAM (instead of 2 Kbytes) as well as an on-chip floating-point unit, which was well described in Homewood et al.[1] Except for access time, no difference exists between internal and external memory in either transputer. The total memory space for the T414/T800 is 4 Gbytes, although the design philosophy is that transputers do not need abundant memory. Rather more important is that a number of transputers communicate through their four dedicated serial channels at speeds of up to 20 Mbits per second. Because the address and data buses are multiplexed, the resulting package contains only 68 pins.

Downloading a bootstrap program into internal or external memory can be accomplished through the serial links. Hence, a minimum, additional transputer can consist of a single chip.

0272-1732/89/0600-0060$01.00

The T212 is the 16-bit version of the transputer. A word is 32 bits wide on the T414/T800 and 16 bits on the T212. To make the T212 architecture and instruction set compatible with those of the T414, the assembler refers to a bit count of 16 instead of 32 and multiplies by two instead of four in terms of bytes. This approach is quite different from the process involved with M68000/MC68020 compatibility, for instance, because the data structures change rather than the number of transfer cycles.

The M212 is an especially programmed disk processor. It contains a modified T212 16-bit, on-chip memory, and two byte-wide bidirectional ports. Disk-interface hardware provides for a standard floppy or Winchester disk interface. A set of procedures in ROM performs basic disk accessing.

The T414 is a reduced instruction set computer (RISC) primarily because its architecture does not contain the standard registers and the more-or-less orthogonal operations of conventional microprocessors. Its instruction set is simple and efficient; the instruction cycle is very short.

Inmos designed the concurrent language Occam[1,2] for the transputer to have both high- and low-level facilities. Occam specifically programs a set of communicating processes that take place between transputers via channels. The instruction set was designed to promote Occam compiler efficiency.

Registers, memory, and communications

The architecture of the T414 is rather simple and borrows architectural ideas from Texas Instrument's TMS9000 microcomputer and the old Hewlett-Packard calculators.

Registers. As summarized in Figure 2, the T414 processor consists of an instruction pointer I (the usual

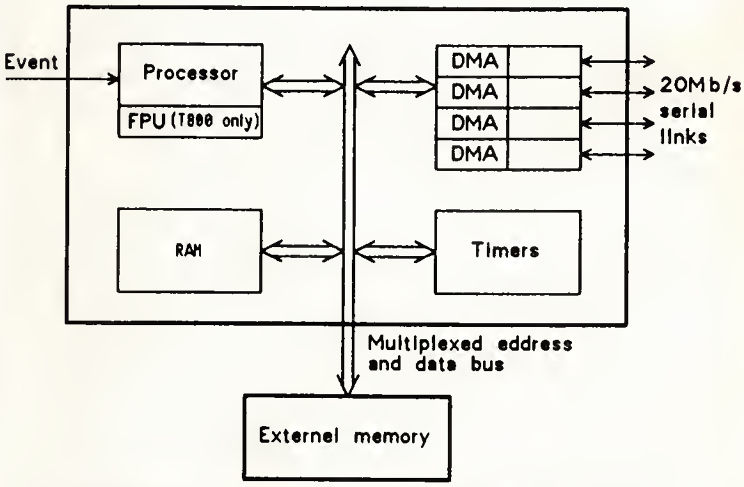


Figure 1. Block diagram of the T414/T800 transputer.

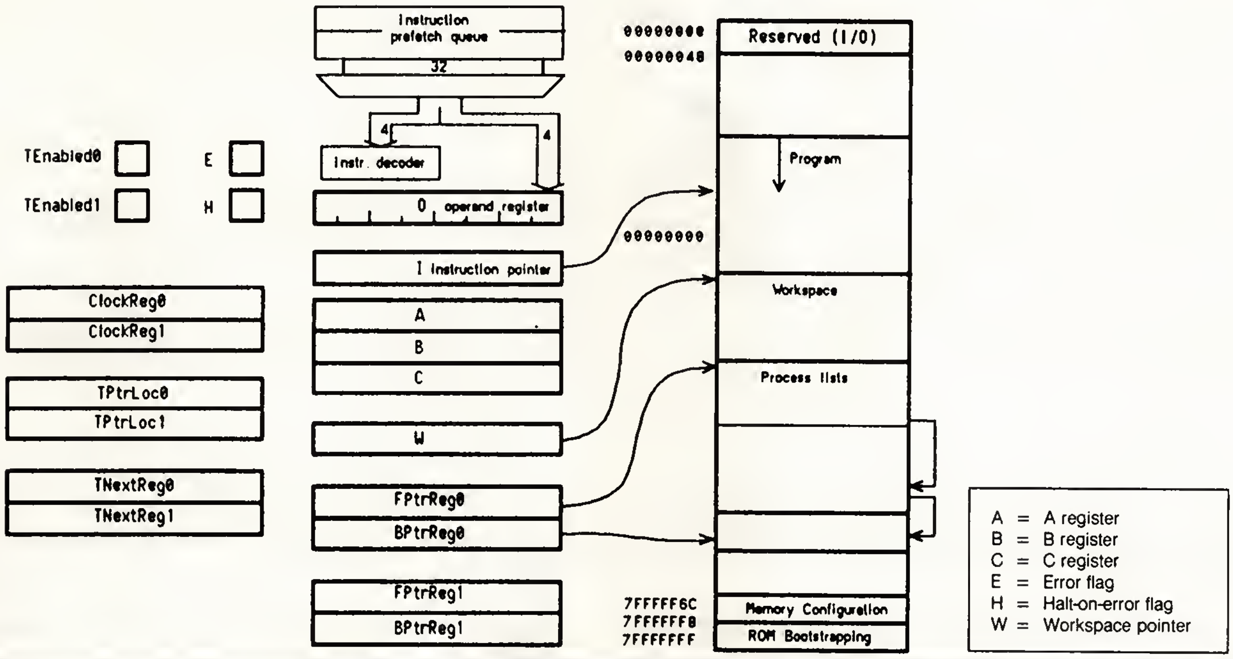


Figure 2. Processor model.

The T414 instruction set is simple and efficient.

program counter) and an invisible instruction-fetch buffer of two 32-bit registers. The processor extracts 8-bit instructions from this buffer to prepare the contents of the 32-bit operand register O. The usual microprocessor instruction register splits into the operand register O and the 4-bit function code of the last byte of each instruction, as shown later.

Memory. The 32-bit workspace pointer W can point anywhere in memory, making context switching easy and fast. A single-byte instruction accesses the first 16 locations. Three 32-bit registers A-C form an arithmetic evaluation stack. They should not be seen as a set of independent registers, which is the way they usually function on many other microprocessors. The compiler allocates room for user registers inside the workspace of a process. Parameters, even temporarily, are never kept in the evaluation stack. Instructions that do not use registers A, B, or C, like Jumps, leave these registers in some unpredictable state. As detailed later, the usual condition status register (CSR or flags) does not exist. Overflow indications or results of a comparison remain in register A. An error flag E and a halt-on-error flag H exist for handling overflows, but have no correspondent in standard microprocessors.

The processor maintains two linked scheduling lists for simulating parallel operations. Four registers specifically form these linked lists. Of the two registers for high-priority processes, one (FPtrReg0) points to the front of the list and the other (BPtrReg0) to the back. Two similar registers (FPtrReg1 and BPtrReg1) perform low-priority processes. The process-scheduling registers are detailed later.

The local timing operations on the transputer employ six registers:

- two clock (timer) registers (ClockReg0 and ClockReg1—one for each priority level),
- two registers pointing to the first items on the two priority timer queues (TPtrLoc0 and TPtrLoc1), and
- two registers indicating the time of the first event to occur (TNextReg0 and TNextReg1—again, one for each priority level).

Two bits indicate whether there is anything on either of the timer queues (TEnabled0 and TEnabled1). The high-priority clock increments every 1 μs, the low-priority clock every 64 μs.

Communications. A channel provides a communication path between two processes. Single words in memory (internal channels) implement channels between processes executing on the same transputer. Point-to-point links (external channels) implement channels between processes executing on different transputers. The compiler allocates the memory location for internal communications. Four external channel locations are reserved at what is considered as the bottom of transputer memory (H'80000000 to H'8000001C, where H' indicates HEX, or hexadecimal). (See Reserved I/0 area in Figure 2.)

The implementation of external communications uses three invisible, separate registers to support autonomous DMA, which frees the processor for other work. These link registers hold

- the count of the number of bytes to be transferred,
- a pointer to the location in memory (to input or output), and
- a pointer to the workspace of the process.

When control is transferred to the link registers, the process is removed from the schedule. Once the communication is complete, the link interface signals the processor to add the process to the end of the list of active processes.

Addressing modes

As in all RISCs, the addressing modes are very simple. The Common Assembly Language for Microprocessing (CALM) notations[3,4] appear here in addition to the Inmos notations.[5,6] Those Inmos notations that differ from CALM appear in italics.

The main conventions of CALM are the use of

- A, B, etc., for register names (reserved symbols);
- ADDR expressions (numbers) for memory or I/O addresses;
- {A}, {ADDR} for contents of registers A or ADDR (corresponding to a value or a memory address);
- {A} + DISP memory location for the address that is the sum of the contents of register A and the value DISP; and
- {A + } notations to indicate that the contents of register A are incremented after the transfer of the location pointed to by register A.

Immediate mode. The immediate-addressing mode (named load constant by Inmos) takes the value prepared by the instruction in the operand register. Only 32-bit transfers are possible, but short instructions exist to move 4, 8, 12, . . . , 32-bit values zero- or sign-extended to 32 bits. The sole destination is the evaluation stack. For instance, the following instructions (in the following examples, the CALM notation appears first, then the Inmos notation with an explanation of the instruction):

Table 1.
Comparison of CALM and Motorola notations.

CALM		M68000	
MOVE.32	{{A6} + DISP1 * 4} + DISP2 * 4,D0	MOVE.32	([DISP1 * 4, A6,ZA0 * 1,DISP2 * 4),D0
MOVE.32	{A6} + DISP1 * 4,A6	MOVE.32	(DISP1 * 4,A6),A6
MOVE.32	{A6} + DISP2 * 4,D0	MOVE.32	(DISP2 * 4,A6),D0

MOVE.32 #3,A-B-C (*ldc 3*, load constant)

copy the value 3 into register A, A copies into B, and B copies into C.

CALM refers to the Inmos load-pointer mode with the term "generalized immediate addressing" (other microprocessors use the term "load address") as follows:

MOVE.32 #{A} + DISP,A(*ldnlp disp*, load nonlocal pointer, that is, load pointer from memory outside the workspace).

Notice that this instruction addresses a word that is aligned with a multiple of two in the T212 and a multiple of four in the T414. It is equivalent to an ADD.32 #DISP,A instruction and loads into A the contents of register A (considered as a pointer to a table), plus the word displacement DISP.

Relative mode. Absolute addressing does not exist on the transputer, except for special instructions in which the Inmos notation is implied for timer, communication, and scheduling locations. Relative addressing is only available for Jump instructions and guarantees the relocatability of modules. Displacement can be a 4, 9, 13, . . . , 32-bit signed value. For instance,

JUMP ADDR (*j addr*, jump to address *addr*)

jumps to the location ADDR; the value coded in the instruction is the displacement from the instruction pointer.

Register-indirect mode. Register-indirect addressing is called local when it refers to the workspace register W, and nonlocal when it refers to the A register as a base pointer. For instance, because DISP is a word displacement, one can write

MOVE.32	A,{W} + DISP	(*stl disp*, store local)
MOVE.32	{A} + DISP,A	(*ldnl disp*, load nonlocal)
MOVE.8	{A},A	(*lb*, load byte)
MOVE.32	{A} + 4 * {B},A	(*wsub*, word subscript)

The double-indirect mode, named static chain by Inmos, can be written in CALM:

MOVE.32 {{W} + DISP1} + DISP2,A

This mode is implemented by two transputer instructions (preceded by their CALM equivalents).

MOVE.32	{W} + DISP1,A-B-C	(*ldl disp1*, load local)
MOVE.32	{A} + DISP2,A	(*ldnl disp2*, load nonlocal)

Table 1 provides equivalent M68000 notations for the previous three CALM instructions. Transputer registers W and A are mapped onto MC68020 registers A6 and D0.

Data types

The T414 supports only 8- and 32-bit integers. The 8-bit numbers are extended to zero when used as 32-bit values. The T800 also supports 32- and 64-bit floating-point numbers. The 32-bit addresses and integers are signed (two's complement). Address 2^{31} = H'80000000 (*MostNeg*, most negative) is reserved for communication and scheduling. Addresses of 32-bit words (data, pointers) must be word aligned in memory; the last two bits of these addresses, named a byte selector, must be zero.

Programming philosophy

Due to the architecture and reduced instruction set of the transputer, its programming philosophy is quite different from that of other 32-bit processors like the DEC VAX, Motorola M68000, National Semiconductor NS32000, or Intel iAPX-386. Transputer workspace, stack, and overflow handling require a complete change in thinking for the programmer who is accustomed to 8-, 16-, and 32-bit sequential processors.

Workspace. Each parallel process has a workspace associated with it, that is, a block of 32-bit words in memory. Register W points to the beginning of this block during the execution of the process. The local variables and the return address are both reached by positive offsetting. Negative offset addresses are used for the timer and communications. The workspace address and its priority level, given in the process descriptor, completely identify a process. The priority of a process is stored in the least significant bit (LSB) of the workspace address.

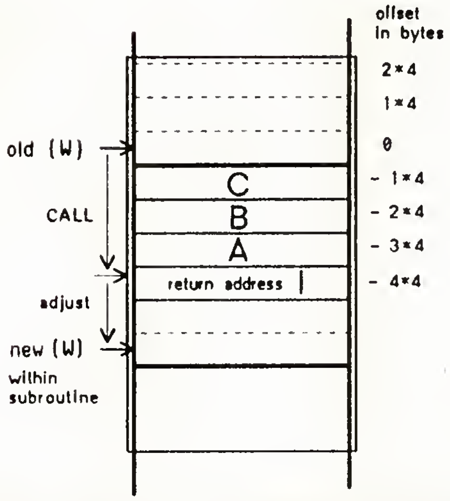


Figure 3. Workspace and subworkspace during a subroutine.

Calling a subroutine creates a subworkspace. The system automatically reserves four words—the return address (instruction pointer I) and the three evaluation stack registers (A, B, and C), as shown in Figure 3. Adjust instructions allow space to be allocated for variables. The system reserves locations above the main workspace for process scheduling.

Evaluation stack. Load instructions prepare the evaluation or operand stack. The register pair A,B can be exchanged by using the transputer Reverse (*rev*) instruction. Two-operand Arithmetic and Logic instructions use registers A and B, with the result placed in A. The contents of register C usually move into B; the new contents of C remain undefined.

Some instructions take their operands from the stack. Move Block instructions, for instance, consider C as the source pointer, B as the destination pointer, and A as the block-length counter.

Overflow handling. Unlike other microprocessors, the transputer has no equal or carry flags. Rather, at the end of some operations, register A is loaded with a single bit. The meaning of this bit corresponds to one flag. The error-flag bit is set by several instructions when the result overflows (see section on arithmetic instructions).

Concurrent processes

The transputer contains a real-time, hard-wired kernel. A process is defined by its workspace address. Workspaces are linked to form two lists of waiting processes (priority 0 = high and 1 = low). Special instructions (shown later) allocate and link processes.

Process scheduling. Dedicated registers in the processor point to the front and back of the active process lists, as shown in Figure 4. One can see four processes waiting to be executed: three are high priority and one is low. FPtrReg0 points to the workspace of the process (value W1) at the front of the high-priority list (a). The list pointer of W1 (noted W1-2 and found at address {FPtrReg0} − 2 ∗ 4) points to the next process' workspace W3 (b). The word at address W1-1 contains

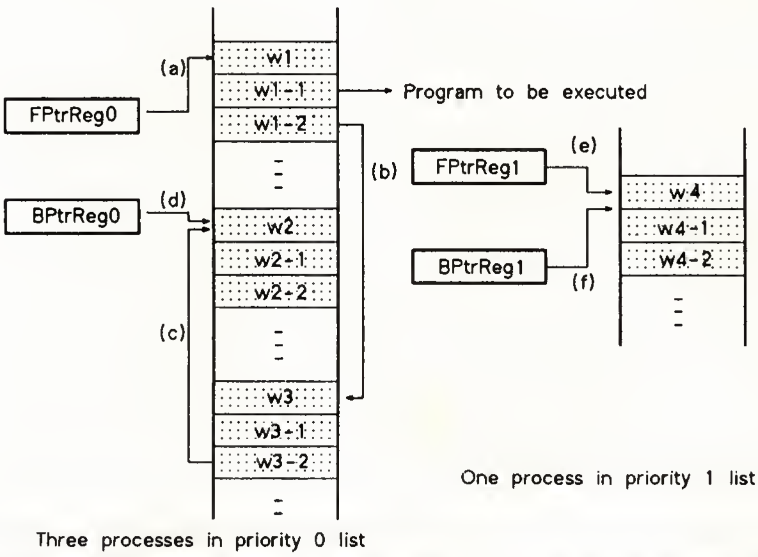


Figure 4. Typical process lists.

READER-SERVICE CARD

IEEE MICRO

June 1989 issue (card void after December 1989)

Name ______________________

Title ______________________

Company ______________________

Address ______________________

City ______________ State_____ ZIP ________

Country ____________ Phone (_____) ____________

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest

(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150	159	168	177	186
151	160	169	178	187
152	161	170	179	188
153	162	171	180	189
154	163	172	181	190
155	164	173	182	191
156	165	174	183	
157	166	175	184	
158	167	176	185	

Product Information 1

(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

READER-SERVICE CARD

IEEE MICRO

June 1989 issue (card void after December 1989)

Name ______________________

Title ______________________

Company ______________________

Address ______________________

City ______________ State_____ ZIP ________

Country ____________ Phone (_____) ____________

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest

(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150	159	168	177	186
151	160	169	178	187
152	161	170	179	188
153	162	171	180	189
154	163	172	181	190
155	164	173	182	191
156	165	174	183	
157	166	175	184	
158	167	176	185	

Product Information 2

(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

❑ **YES**, sign me up!

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$18** for a year's subscription (six issues).

Society: ______________ IEEE membership no.: ______________

Society members: Subscriptions are annualized. For orders submitted March through August, pay half the full-year rate ($9) for three bimonthly issues.

❑ **YES**, sign me up!

If you are a member of ACM, ACS, BCS, IEE (UK), IEEE (but not a member of an IEEE society), IECEJ, IPSJ, NSPE, SCS, or other qualified professional technical society, pay the sister-society rate of only **$33** for a year's subscription (six issues).

Organization: ______________ Membership no.: ______________

Full Signature ______________ *Date*

Name ______________

Street ______________

City ______________

State/Country ______________ *ZIP/Postal Code*

❑ Payment enclosed

❑ Charge to ❑ Visa ❑ MasterCard ❑ American Express

Charge-card number

— — — — — — — — — — — — — — — —

Expiration date

— —   — —

Month   *Year*

Prices valid through 12/31/89

6/89 MICRO

Charge orders also taken by phone:
(714) 821-8380 8 a.m. to 5 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-2578

Editorial comments

I liked:

I disliked:

I would like to see:

For reader-service inquiries, see other side.

PO Box is for reader-service cards only.

PLACE
POSTAGE
HERE

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508

Editorial comments

I liked:

I disliked:

I would like to see:

For reader-service inquiries, see other side.

PO Box is for reader-service cards only.

PLACE
POSTAGE
HERE

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

IEEE COMPUTER SOCIETY
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9804
USA

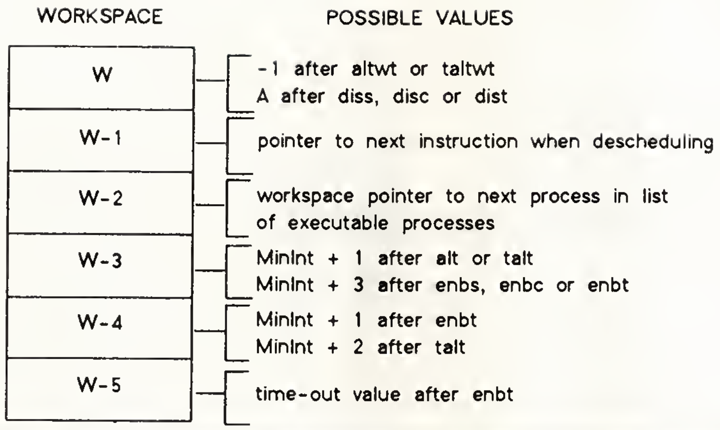


Figure 5. Workspace locations for the Alternative (ALT) instruction.

a pointer to the first instruction to be executed when the process is executed. BPtrReg0 points to the workspace of the last process (d) as does W3-2. As only three processes occupy this list, the process's workspace pointed to by (c) and (d) is the same (W2). For the single low-priority process W4, both front and back pointers (FPtrReg1 and BPtrReg1) point to the same process' workspace W4 (e,f).

A new process is placed at the end of the high- or low-priority list. When the current process is removed from the schedule, it is placed at the back of the list by updating BPtrReg, and the process at the front of the list is executed.

A high-priority process always takes precedence over a low-priority process. A low-priority process is removed from the schedule when a high-priority process becomes available for execution. When no more high-priority processes exist, the low-priority process continues to execute.

Seven locations near the bottom of the transputer's memory map hold the state of a preempted, low-priority process. One process, at most, can be preempted at one time to prepare seven, 32-bit memory locations.

Alternatives. The Occam Alternative (ALT) instruction is a very powerful construct that allows choices between a number of branches. The testing of inputs and/or Boolean conditions—with the possible inclusion of timers—provides the information to make these choices. The transputer contains 11 instructions to realize this construct. Specific locations in the workspace of a process implement this instruction. The workspace notations shown in Figure 5 correspond to any of the four workspaces shown in Figure 4. W, W-1, and W-2 have the same meaning as W1, W1-1, and W1-2 for process scheduling. W-3 signals a satisfied alternative, with MinInt + 3 representing the value true and MinInt + 1 representing the value false (MinInt, or the Minimum Integer value is equal to 2^{31} on the T414 and 2^{15} on the T212). W-4 is used if a timer is included for a time-out. It signals an enabled timer, with MinInt + 1 as true and MinInt + 2 as false. Finally, W-5 contains the earliest time a time-out can occur (if one is specified within the ALT instruction). Figure 5 summarizes these values. Workspace content depends upon the instructions that are executed. The instructions mentioned in Figure 5 are explained later.

Timer queue. The Timer Input (*tin*) instruction can suspend processes on a transputer until a specified time. The suspended processes are held in one of two linked lists (one for each priority level). Each list has a slightly different structure, as shown in Figure 4. W-4 in Figure 5 contains a pointer to the next process in the list, except for the last process, in which W-4 contains MinInt. Location W-5 holds the designated time for awakening the process. The process order in the list is determined by the value in W-5. W-3 always contains the value MinInt + 2 for a process on a timer queue. Locations W, W-1, and W-2 are used as previously described.

Structure

The basic transputer instruction set consists solely of 1-byte instructions. To obtain instructions that look similar to those of the usual microprocessor, one needs several transputer instructions. For instance, the previously mentioned Move Block instruction starts with its operand in the workspace. It takes three instructions to move the registers onto the evaluation stack before the transputer Move Block instruction can occur. In the best case, if the operands are initially in the first sixteen, 32-bit word locations of the workspace, this set of instructions takes 5 bytes. In the worst case, it can take up to 26 bytes.

T414 instruction set

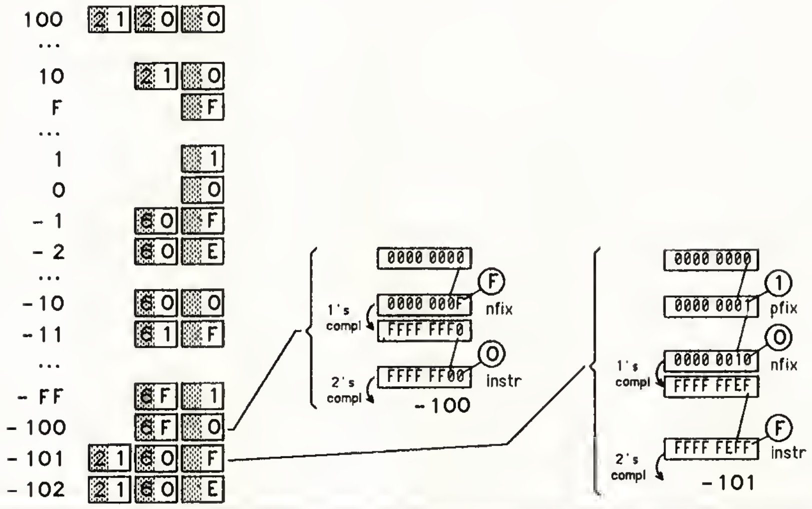


Figure 6. Coding of several positive and negative integers.

This difference in length is due to the particular encoding of the offsets and immediate values. As mentioned, a basic instruction consists of a 4-bit operation code and a 4-bit operand, which are loaded into the operand register. The operand register is cleared at the end of all instructions. Values larger than 4 bits can be prepared in the operand register by using two basic instructions, Prefix (*pfix*) and Negative Prefix (*nfix*). The Prefix instruction loads its 4-bit operand into the operand register and then shifts the contents of the operand register O to the left by 4 bits (Figure 2). This procedure allows the preparation of instructions that require a 4-, 8-, 12-, . . . , 32-bit positive value or displacement. Negative values or displacements must be prepared by appropriate use of the Negative Prefix instruction, which loads the 4-bit operand in the low bit of the operand register, complements the complete register, and then shifts 4 bits to the left.

Figure 6 illustrates how the Occam compiler uses this process to encode operands between the hexidecimal values 100 and −102 (instruction codes or prefixes are shaded). For instance, value −100 is encoded H'6FX0, where 6 is the negative prefix, and F is a first digit loaded into the operand register (and is immediately complemented). X (shown as a shaded blank) is the 4-bit code of the instruction, and 0 is the second digit that must be shifted into the operand register to obtain the result. Students can gain from the exercise of writing the routine to prepare the correct digits according to the number.

As an example of this mechanism within instructions, let us consider the Jump instruction, the 4-bit code of which is zero (Figure 7). If the displacement is less than 4 bits and is positive, a single byte is sufficient to code the Jump instruction. The value added to the instruction pointer I is the contents of the operand register that has been loaded with the 4-bit value.

Using a Prefix (*pfix* with a 4-bit code 2) or Negative Prefix (*nfix* with a 4-bit code 6) instruction before the basic Jump instruction allows a positive 8-bit operand, or a negative 4-bit one. The Prefix instruction can be used as many times as required, as shown in Figure 7. This procedure provides rather short instructions when the operands are short. However, when 32-bit operands are used—which is rarely—the instructions are not very efficient.

Prefix instructions also extend the instruction set up to 2^{32} instructions. The T414 implements less than 512 instructions by using one Prefix instruction and a special Operate (*operate*) instruction coded with the hexadecimal number F. This coding means that the operand register now contains the instruction code. These instructions have no explicit operand.

Hence, transputers have two basic Prefix instructions, 13 basic instructions that use the operand register as an immediate value or displacement, and one Operate instruction. This arrangement allows the building of 16 short instructions from the 4-bit associated value. It also allows 512 instructions that have the format $2n_1Fn_0$ or $6n_1Fn_0$, where 2 and 6 are the code of the positive and negative Prefix instructions, and F is the Operate instruction.

CALM and Inmos notations

These notations complement each other. CALM describes exactly what each instruction does at the register transfer level, while the Inmos notation emphasizes the use of the instruction when it is handling the usual data structures. However, CALM has been designed for microprocessors with several registers and memory spaces and a rather orthogonal set of notations. It is not very well suited for stack machines, since CALM expresses everything the instruction does (excluding action on flags). The evaluation stack of the T414 is a region in which several transfers can occur simultaneously. The transfer rules (that is, which registers are updated or modified) are sometimes not in-

tuitive. CALM mentions only the major operands; secondary registers that are modified are shown between brackets, as are the flags.

The Inmos notations do not explicitly specify the transputer registers. The evaluation stack and the workplace pointer are never mentioned. These instructions are hence very short and easy to enter, but not very readable to a programmer unfamiliar with the processor.

Inmos sometimes uses condensed notations that correspond to several instructions. For instance, the Inmos notation for the Move Block instruction, assuming the three operands are at offsets SOURCE, DEST, and LENGTH from the top of the workspace, is *move*. The Occam notation for this is v1 : = v2, which means that destination vector v1 becomes vector v2. This notation corresponds to

MOVE.32	{W}+SOURCE,A-B-C	(*ldl source,* load local)
MOVE.32	{W}+DEST,A-B-C	(*ldl dest*)
MOVE.32	{W}+LENGTH,A-B-C	(*ldl length*)
REP	{A} MOVE.8 {C+},{B+}	(*move*)

where REP {A} MOVE.8 {C+},{B+} means repeat the Move Byte instruction the number of times indicated by the contents of register A.

Homewood et al.[1] uses the Move instruction in discussion of Move2d, which moves a block of data corresponding to a rectangle on a screen. In addition to the Move instruction, the compiler generates tens of instructions for preparing the workspace and giving the control back to the calling module.

Instruction set

We present the T414 instructions in the order used by most microprocessor manufacturers. Inmos has a slightly different order, since it tends to group the instructions that use the evaluation stack, the workspace, and the channels.

The remaining figures in this article present (from left column to right)

- the instruction code with shaded codes or prefixes,
- the Inmos notation for the same code,
- the Inmos mnemonic instruction and its explanation,
- the CALM mnemonic and operand expressions that clearly show the addressing modes, and
- a list of the registers that are modified.

[A] means that register A is modified according to the logic of the instruction. [A: = B] means that register A is loaded with the contents of B. When a register is left undefined by an instruction, the symbol ϕ appears (for example, [Aϕ] stands for A undefined).

Jump instructions. These instructions cause a deviation from the normal sequential flow (see Figure 8 on the next page). Surprisingly, registers A, B, and C are modified by a Jump or Call instruction. The Jump Relative and Conditional Jump instructions (*j* and *cj*) are relative to the instruction pointer. The Conditional Jump instruction is only performed if the contents of register A = O. Note that there is no flag register; the system checks the condition on the top of the stack instead.

The Loop End instruction (*lend*) controls replications. A loop is controlled by two words in memory pointed to by register B (addresses {B} and {B}+4). The first word is the control variable incremented when it exits the loop; the second word is the number of iterations that remain. Register A holds the number of bytes between the start of the next instruction and the beginning of the loop. The Loop End instruction checks the number of iterations, {B}+4. When the number is greater than 1, the instruction jumps back to the start of the loop, increments the control variable, and decrements the iterations. When the number is not greater than 1, the number of iterations is decremented, and execution is passed to the next instruction.

The Call and Return instructions (*call* and *ret*) implement procedures. The Call instruction pushes the instruction pointer and the evaluation stack onto the

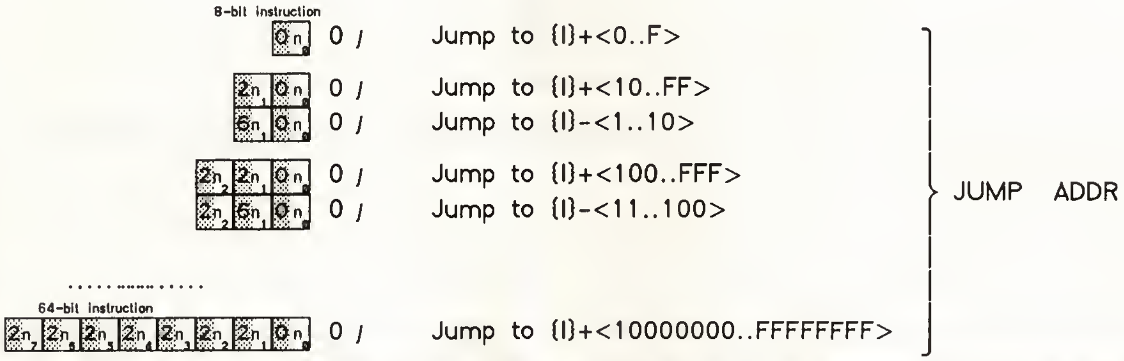


Figure 7. Jump instruction with different operand lengths.

T414 instruction set

Encoding	Code	Mnemonic	Operation	Operands	Registers	Description
..... 0n_0	0	*j*	JUMP	ADDR	[AØ,BØ,CØ,EØ]	jump relative (offset in bytes, ADDR = (I) + ...n_0)
..... An_0	A	*cj*	JUMP,AEQ	ADDR	[none or A:=B,B:=C,CØ]	conditional jump (If A≠0, A:=B,B:=C, otherwise jump with A,B,C unchanged)
2 2 F 1	F/21	*lend*	DJ,PL	{B}+4,{I}-{A}	[AØ,BØ,CØ,EØ]	loop end (if (B)+4 > 1; jump to (I)-(A), increment (B) and decrement (B)+4, otherwise decrement (B)+4 and continue)
..... 9n_0	9	*call*	CALL	ADDR	[W,A:=I,BØ,CØ]	call (SUB.32 #4*4,W; MOVE.32 I,(W); MOVE.32 A,(W)+4; MOVE.32 B,(W)+8; MOVE.32 C,(W)+12, see fig 2)
2 2 F 0	F/20	*ret*	RET		[W]	return (MOVE.32 (W),I; ADD.32 #4*4,W)
F 6	F/6	*gcall*	EX	A,I	[A]	general call

Figure 8. Jump instructions.

Encoding	Code	Mnemonic	Operation	Operands	Registers	Description
..... 4n_0	4	*ldc*	MOVE.32	#VAL,A-B-C	[A,B:=A,C:=B]	load constant (VAL = ...n_0)
2 4 F 2	F/42	*mint*	MOVE.32	#MinInt,A-B-C	[A,B:=A,C:=B]	minimum integer MinInt=-2**31
..... 7n_0	7	*ldl*	MOVE.32	{W}+DISP,A-B-C	[A,B:=A,C:=B]	load local (DISP = 4*...n_0)
..... Dn_0	D	*stl*	MOVE.32	A,{W}+DISP	[A:=B,B:=C,CØ]	store local (DISP = 4*...n_0)
..... 1n_0	1	*ldlp*	MOVE.32	#{W}+DISP,A-B-C	[A,B:=A,C:=B]	load local pointer (DISP = 4*...n_0)
..... 3n_0	3	*ldnl*	MOVE.32	{A}+DISP,A	[A]	load non local (DISP = 4*...n_0)
..... En_0	E	*stnl*	MOVE.32	B,{A}+DISP	[A:=C,BØ,CØ]	store non local (DISP = 4*...n_0)
..... 5n_0	5	*ldnlp*	MOVE.32	#{A}+DISP,A	[A]	load non local pointer (equivalent to ADD.32 #DISP,A)
F 1	F/1	*lb*	MOVE.8	{A},A	[A]	load byte (A is extended with zeros, DISP = ...n_0)
2 3 F B	F/3B	*sb*	MOVE.8	B,{A}	[A:=C,BØ,CØ]	store byte
F F	F/F	*outword*	MOVE.32	A,4*{B}	[AØ,BØ,CØ,EØ,{W}Ø]	output word (B points to a channel)
F E	F/E	*outbyte*	MOVE.8	A,{B}	[AØ,BØ,CØ,EØ,{W}Ø]	output byte (B points to a channel)
2 1 F B	F/1B	*ldpi*	MOVE.32	#{I}+{A},A	[A]	load pointer to instruction (ADD.32 I,A) (offset in A measured in bytes)
F 2	F/2	*bsub*	MOVE.32	#{A}+{B},A	[A,B:=C,CØ]	byte subscript (ADD.32 B,A)
F A	F/A	*wsub*	MOVE.32	#{A}+4*{B},A	[A,B:=C,CØ]	word subscript (ADD.32 B*,A

Figure 9. Transfer instructions.

workspace (as in Figure 3). The Return instruction restores the instruction pointer and adjusts the workspace pointer to deallocate the four locations. Registers A, B, and C are neither restored nor modified. The General Call instruction (*gcall*) interchanges register A and instruction pointer I. This interchange enables the entry point of a procedure to be computed by one or more instructions in register A before the General Call instruction is executed.

Transfer instructions. These instructions enable variables and pointers to be fetched or stored (Figure 9). There are two basic types: local (with respect to the workspace pointer) and nonlocal (with respect to regis-

24 FA	F/4A	*move*	REP {A}	MOVE.8 {C+},{B+}	[AØ,BØ,CØ]
		move message (Occam v1:=v2, block is (A) byte long)			
F7	F/7	*in*	REP {A}	MOVE.8 {B},{C+}	[AØ,BØ,CØ,EØ]
		input message (Occam: chan ? var)			
FB	F/B	*out*	REP {A}	MOVE.8 {C+},{B}	[AØ,BØ,CØ,EØ]
		output message (Occam: chan ! exp)			

Figure 10. Move Block instructions.

F0	F/0	*rev*	EX.32	A,B	[A,B]
		reverse			
23 FC	F/3C	*gajw*	EX.32	A,W	[W,A]
		general adjust workspace			

Note: .32 means 32-bit unsigned content .A32 means 32-bit signed (2-s complement) content

23 FA	F/3A	*xword*	CONV	B,A.A32	[A,B := C,CØ]
		extend to word (if B < A, A := B, otherwise A := (B - 2*A))			
21 FD	F/1D	*xdble*	CONV	A.A32,BA.A64	[B := -1 if A<0,C := B] [B := 0 if A⩾0,C:=B]
		extend to double (extend A into A and B)			
24 FC	F/4C	*csngl*	CONV	AB.A64,A.A32	[B := C,CØ,E]
		convert single (if (A<0 and B≠-1) OR (A⩾0 and B≠0) E := 1, otherwise unchanged)			
23 F4	F/34	*bcnt*	SL.32	#2,A	[A]
		byte count (MUL.32 #4,A)			
23 FF	F/3F	*wcnt*	ASR.32	#2,A	[A,B,C := B]
		word count (B gets the two low bits of A) (DIV.32 #4,A)			

Figure 11. Exchange and Format Conversion instructions.

ter A). Many of these instructions are similar to those found on standard microprocessors. For instance, the Load Constant instruction (*ldc*) pushes a constant onto register A. The Minimum Integer instruction (*mint*) pushes the most negative address MinInt onto register A. The value of *MinInt* is equal to H'80000000 on the T414.

The Load Local, Load Local Pointer, and Store Local instructions (*ldl, ldlp,* and *stl*) access locations relative to register W. A local variable can be placed on the evaluation stack by the Load Local instruction. Its address is placed on the stack by the Load Local Pointer instruction. The Store Local instruction stores the value at the top of the stack back into the variable. The instructions Load Nonlocal, Store Nonlocal, and Load Nonlocal Pointer (*ldnl, stnl,* and *ldnlp*) are similar to the local instructions except they are relative to register A, not register W, which allows a level of indirection. In the case of the Store Nonlocal instruction, the value in register B is stored at the address found in register A. Instructions Load Byte and Store Byte (*lb* and *sb*) are used for byte transfers. In multiple transfers, Output Word and Output Byte instructions (*outword* and *outbyte*) communicate a single word and a single byte in register A through the channel pointed to by register B (referred to as the B channel), using the first word of the workspace. The Load Pointer to Instruction instruction (*ldpi*) pushes the address of a location in a program onto the stack.

The Byte Subscript instruction (*bsub*) returns the sum of B and A register contents into register A without any overflow check. The Word Subscript instruction (*wsub*) does the same, but first they both multiply the contents of register B by four (on the T414). These two instructions interpret register A as the address of the beginning of a data structure. The result of these instructions leaves the address of the byte—which is the byte or word content of register B (referred to as B bytes or words) from the beginning of the structure—in register A.

Move Block instructions. These instructions allow a block of elements to be moved from one area to another (Figure 10). The Move instruction described earlier moves the number of data bytes found in the A register starting at the address pointed to by register C and finishing at the address pointed to by register B (the blocks must not overlap). The In and Out instructions (*in* and *out*) communicate data. The In instruction transfers a block of bytes from the B-channel address to the address pointed to by register C. The Out instruction transfers A bytes from the address pointed to by register C to the B-channel address. The processes at either end of a communication should have the same value in register A.

Exchange and format conversion instructions. These instructions change the format of data and exchanging registers (Figure 11). The Reverse instruction exchanges the contents of registers A and B. The General Adjust Workspace instruction (*gajw*) exchanges the W and A registers, allowing dynamic allocation of workspaces as

Code	Opcode	Mnemonic	CALM	Operands	Registers	Description
.....8n_0	8	*adc*	ADD.A32	#VAL,A	[A,E]	add constant (E set if overflow, otherwise unchanged) (VAL = ...n_0)
F5	F/5	*add*	ADD.32	B,A	[A,B := C,CØ,E]	add (E set if overflow, otherwise unchanged)
25F2	F/52	*sum*	ADDn.32	B,A	[A,B := C,CØ]	sum (E not influenced) -- use bsub --
21F6	F/16	*ladd*	ADDC.32	B,A	[A,BØ,CØ,E]	long add A:=A+B+C (Carry is low bit of C, E set is overflow, unchanged otherwise)
23F7	F/37	*lsum*	ADDCn.32	B,A	[A,B,CØ]	long sum A:=A+B+C (Carry is low bit of C, B gets the resulting carry)
FC	F/C	*sub*	ISUB.32	B,A	[A,B := C,CØ,E]	subtract (A:=B-A, E set if overflow, otherwise unchanged)
F4	F/4	*diff*	ISUBn.32	B,A	[A,B := C,CØ]	difference (A:=B-A)
23F8	F/38	*lsub*	ISUBC.32	B,A	[A,BØ,CØ,E]	long subtract A:=B-A-C (borrow is low bit of C, E set if overflow, otherwise unchanged)
24FF	F/4F	*ldiff*	ISUBCn.32	B,A	[A,B,CØ]	long subtract A:=B-A-C (borrow is low bit of C, B gets resulting borrow)
25F3	F/53	*mul*	MUL.32	B,A	[A,B := C,CØ,E]	multiply (E set if overflow, otherwise unchanged)
F8	F/8	*prod*	MULn.32	B,A	[A,B := C,CØ]	product (faster if small operand in A)
23F1	F/31	*lmul*	MULn.32	B,BA	[A,B,CØ]	long multiply (BA:=A*B+C, B gets resulting carry)
22FC	F/2C	*div*	DIV.32	B,A	[A,B := C,CØ,E=1]	divide (quotient; if overflow or A=0, E:=1 and A unchanged, otherwise E unchanged)
21FF	F/1F	*rem*	REM.32	B,A	[A,B := C,CØ,E=1]	remainder (rest; if overflow or A=0, E:=1 and A unchanged, otherwise E unchanged)
21FA	F/1A	*ldiv*	DIV.64	CB,A	[A,B,CØ,E]	long divide (divide CB by A, B gets remainder, E set if invalid, otherwise unchanged)
.....Bn_0	B	*ajw*	ADD.32	#DISP,W	[W]	adjust workspace (DISP = 4*...n_0)

Figure 12. Arithmetic instructions.

well as dynamic switching between existing workspaces. The Extend to Word (*xword*) instruction sign-extends a part-word value to a single-word value. The two operands of the instruction are a part-word in register B and a length specified in register A by the most negative integer representable by the part-word. For instance, to sign-extend a byte in register A to 32 bits, one first loads constant H'80 into the evaluation stack (ldc H'80, code 2840), and then executes *xword* (code 23FA). The Extend to Double instruction (*xdble*) extends the single-length signed value in register A into a double-length signed value in registers A and B (most significant part in B). However, the Convert Single instruction (*csngl*) does the reverse and sets the error flag if an overflow condition exists.

The Byte Count instruction (*bcnt*) multiplies register A by the number of bytes in a word (for example, a 32-bit machine multiplied by 4).

A pointer has two parts: a word address and a byte selector. The number of bits required to represent the byte selector depends on the word length (for example, 1 bit for a 16-bit word length, 2 bits for a 32-bit word length). The Word Count instruction (*wcnt*) decomposes an address into its component word part and byte selector. A word offset goes into register A, a byte selector into B.

Arithmetic instructions. The Add Constant instruction (*adc*) allows a constant to be added to register A and checks for overflow conditions (Figure 12). The other arithmetic instructions can divide into two basic sets: single- and multiple-length (longer than a word) instructions. Multiple lengths are identified by the letter *l* (for long) at the beginning of each instruction. These two sets can further divide into two more sets: those that check for error (overflow)—*add, sub, mul, div, rem, ladd, lsub, lmul,* and *ldiv*—and those that ignore carry and overflow—*sum, diff, prod, lsum,* and *ldiff.* For single-length operations, the left-hand operand is taken into register B and the right-hand operand into A, with the result placed in A. In multiple-length operations, two single-word operands are in registers A and B and a carry (or borrow) operand is in the LSB of C. The CALM mnemonic ISUB means that the operands are inverted with respect to the other microprocessors like the

MC68020, NS32032, or VAX. ISUBB,A computes $A = B-A$ while the usual SUBB,A computes $A = A-B$.

The Adjust Workspace instruction (*ajw*) adjusts the value of the workspace pointer by the number of words in its operand value. A negative number allocates more space.

Compare instructions. These instructions compare and check values on the evaluation stack (Figure 13). Since there is no flag register, the results of these operations are put on the evaluation stack. The value true is represented by 1, the value false by 0. The Equals Constant instruction (*eqc*) sets register A to true if A is equal to the instruction operand. Otherwise, it sets it to false. The Greater Than instruction (*gt*) sets A to true if B > A. If not, it sets it to false. The Check Word instruction (*cword*) checks whether a single word can be represented by a part-word. As with *xword*, the single-length word is in register B and the part-word length in A. The error flag is set if it does not fit. For instance, to check whether the contents of register A is a 12-bit value or not, one pushes H'800 onto A (ldc H'800, code 282040) and executes *cword* (code 25F6). These two instructions do not modify numbers initially in registers A and B.

The Check Subscript instruction (*csub0*) sets the E flag if the unsigned value in register B is greater than or equal to the value in register A. The Check Count instruction (*ccnt1*) sets the E flag if B = 0 or is greater than A (that is, if B is between 1 and the contents of register A). This instruction can check whether the count of an input or output instruction is positive or is less than the number of bytes in the message buffer.

Logical instructions. All logical operations (Figure 14) are performed on the evaluation stack. The instructions Logical And, Logical Or, and Exclusive Or (*and, or,* and *xor*) are bit-wise operations on registers A and B, with the result left in A. The Bit-wise Not instruction (*not*) has only one operand, which is in register A. The Shift Left and Shift Right instructions (*shl* and *shr*) displace the operand in register B by the number of bits

..... Cn0	C	*eqc*	COMP.32	#VAL,A	[A]
		equals constant (if EQ, A:=1, otherwise A:=0)			
F9	F/9	*gt*	COMP.A32	B,A	[A,B:=C,CØ]
		greater than (if B>A A:=1, otherwise A:=0)			
25F6	F/56	*cword*	CHECK,VS	B,A	[A:=B,B:=C,CØ,E]
		check word in B according to format in A (E set if (B≥A) OR (B<(-A)), otherwise unchanged)			
21F3	F/13	*csub0*	CHECK,LS	B,A	[A:=B,B:=C,CØ,E]
		check subsript from 0 (E set if A ≤ B (unsigned integers), unchanged otherwise)			
24FD	F/4D	*ccnt1*	CHECK,EQ	B,A	[A:=B,B:=C,CØ,E]
		check count from 1 (E set if B = 0 OR A < B (unsigned integers), unchanged otherwise)			

Figure 13. Compare instructions.

24F6	F/46	*and*	AND.32	B,A	[A,B:=C,CØ]
		logical and (Occam /\)			
24FB	F/4B	*or*	OR.32	B,A	[A,B:=C,CØ]
		logical or (Occam \/)			
23F3	F/33	*xor*	XOR.32	B,A	[A,B:=C,CØ]
		exclusive or (Occam ><)			
23F2	F/32	*not*	NOT.32	A	[A]
		bitwise not (Occam ~)			
24F1	F/41	*shl*	SL.32	A,B	[A,B:=C,CØ]
		shift left (shift by A locations, if 0 ≤ A < 32, effect undefined otherwise)			
24F0	F/40	*shr*	SR.32	A,B	[A,B:=C,CØ]
		shift right (shift by A locations, if 0 ≤ A < 32, effect undefined otherwise)			
23F6	F/36	*lshl*	SL.64	A,CB	[A,B,CØ]
		long shift left (CB shifted by A, result in BA)			
23F5	F/35	*lshr*	SR.64	A,CB	[A,B,CØ]
		long shift right (CB shifted by A, result in BA)			
21F9	F/19	*norm*	NORM	BA	[A,B,C]
		normalise (shift left until MSB is one, C gets shift count)			

Figure 14. Logical instructions.

T414 instruction set

2 2 F 9	F/29	*testerr*	TCLR	E	[A,B := A,C := B,E := 0]
		test error false and clear (if E=1, A:=0, if E=0, A:=1)			
2 5 F 5	F/55	*stoperr*	STOPERR		[AØ,BØ,CØ,E := 1]
		stop on error (if E=1, MOVE I,(W)+4, process stops, next process selected; if E=0, continue)			
2 1 F 0	F/10	*seterr*	SET	E	[E := 1]
		set error (sets error flag)			
2 5 F 7	F/57	*clrhalterr*	CLR	H	[H := 0]
		clear halt-on-error		(clears H flag)	
2 5 F 8	F/58	*sethalterr*	SET	H	[H := 1]
		set halt-on-error (sets H flag)			
2 5 F 9	F/59	*testhalterr*	TEST	H	[A,B := A,C := B]
		test halt-on-error (A:=1 if H=1, A:=0 if H=0)			
2 2 F A	F/2A	*testpranal*	TEST	FlagAnal	[A,B=A,C=B]
		test processor analysing (A = 1 if analysed, A = 0 if reset)			

Figure 15. Error and Halt instructions.

2 3 F 9	F/39	*runp*	MOVE.32	A,{BPtrRegi-2}	[AØ,BØ,CØ]
			MOVE.32	A,BPtrRegi	
		run process (the process with workspace at A is added to appropriate queue)			
F D	F/D	*startp*	MOVE.32	{I}+{B},{A}-1	[AØ,BØ,CØ]
			MOVE.32	A,{BPtrRegi-2}	
			MOVE.32	A,BPtrRegi	
		start process (process B bytes from *I* with workspace at A is placed on current priority queue)			
F 3	F/3	*endp*		COMP.32 #1,{A}+1	[AØ,BØ,CØ]
				JUMP,NE NEXT$	
				JUMP {A}	
			NEXT$:	DEC.32 {A}+1	
				JUMP {FPtrRegi}	
		end process (if count ≠ 1 process terminates, otherwise new process selected)			
2 1 F 5	F/15	*stopp*	MOVE.32	I,{W}-1	[AØ,BØ,CØ]
			JUMP	{FPtrRegi}	
		stop process (process stops execution, new process selected for execution)			
2 1 F E	F/1E	*ldpri*	MOVE.8	current priority,A	[A,B:=A,C:=B]
		load priority (0 for high priority, 1 for low priority)			
2 1 F 2	F/12	*resetch*	MOVE.32	{A},A	[A]
			MOVE.32	#MinInt,{A}	
		reset channel (if A points to the link channel, the link hardware is reset)			

Figure 16. Process-manipulation instructions.

specified in A, or A bits. Vacated bits are set to zero. Long Shift Left and Long Shift Right (*lshl* and *lshr*) shift A bits the double-length value held in registers B and C (the most significant word, or MSW, in C). The result remains in registers A and B (the MSW in B). The Normalize instruction (*norm*) performs floating-point arithmetic and normalizes a double-length value in registers A and B (the MSW in B). The value in register AB shifts to the left until the most significant bit (MSB) of the value is 1. The number of bits shifted remains in register C. It is better to use a T800 because of the tricky T414 floating-point routines.

Error and Halt flag control. The error flag E must be copied into register A to be checked by software. The Test Error (*testerr*) instruction puts a false value (value 0) onto the evaluation stack (that is, register A) if E is set. Otherwise, it places a true value on the stack and clears the E flag (Figure 15). The Stop on Error (*stoperr*) instruction removes the current process from the schedule if the E flag is set. Other instructions (*seterr, clrhalterr,* and *sethalterr*) set the error flag and clear or set the halt-on-error flag. The Test Halt-on-Error (*testhalterr*) instruction pushes the value of the halt-on-error flag onto the evaluation stack (without any inversion). The Test Processor Analyzing instruction (*testpranal*) tests whether a processor has been analyzed or reset. It puts a true value into register A if the processor is analyzed, a false value if it is reset.

Process manipulation. As mentioned, a transputer can run more than one process by using the on-chip kernel. This set of instructions (Figure 16) adds or removes processes from the schedule when operating in this mode. The Load Priority instruction (*ldpri*) loads the priority of the current process onto the evaluation stack. The Start Process and End Process instructions (*startp* and *endp*) perform process initiation and termination. The Start Process instruction initiates a new process (that is, initializes a new workspace) at the current priority level. Register B contains the offset from the current instruction to the new process. Register A has the address of the workspace of the new process. The End Process instruction synchronizes the termination of the current process. The process that started the other concurrent processes keeps a count in the address {A} + 1 of the number of processes still to terminate. Register A contains the workspace address of the initiating process.

The End Process instruction also checks the number of concurrent processes still to terminate. If just one exists, control passes back to the initiating process (workspace pointed to by register A). If more than one process is occurring, the number of concurrent processes is decremented at the address {A} + 1, and the process at the front of the schedule list is taken. The Run Process instruction (*runp*) starts a process. Register A contains the process descriptor of an existing process, which should point to a workspace in which location W-1 (see Figure 5) contains the value to be loaded into instruction pointer I when the process is scheduled. The priority level is determined by the LSB of register A and can be set before the Run Process instruction is executed. The Stop Process instruction (*stopp*) halts the current process that is saving instruction pointer I in the workspace. The Reset Channel instruction (*resetch*) allows a channel to be reset. Register A contains the address of the channel. The channel process word is returned to register A and reset to the value *MinInt*. If the value returned to register A is equal to *MinInt*, the communication finishes successfully. If not, register A contains the process identification number of the channel. This identification can be used by a Run Process instruction to restart the process. If register A was pointing to a link, that hardware would be reset.

Alternatives. Several instructions implement the Occam ALT constructs and its associated instructions on the transputer (Figure 17). An alternative construct selects one of its component alternatives. The selection of the alternative is performed by

- an Alternative instruction (*alt*),
- a sequence of Enable instructions (one for each alternative),
- an Alternative Wait instruction (*altwt*),
- a sequence of Disable instructions (one for each alternative), and
- an Alternative End instruction (*altend*).

The first ready alternative to be disabled is selected. The Enable instructions (*enbs* or *enbc*) are conditional instructions that test a Boolean value in register A named process guard (condition for execution). These instructions also set the process-ready flag in the workspace. The channel component for an enable channel is passed into register B. If register A is true, the guard is enabled. The Disable instructions (*diss* and *disc*) have the offset from the start of the instruction following the Alternative End to the start of the code for that branch of the alternative in register A. The tested condition is in register B, with the channel component in C. The instructions return a true value into register A if that branch of the alternative is the one that has been selected. We provide more details and examples on alternatives and parallel processes in an upcoming publication.[7]

2 4 F 3	F/43	*alt*	MOVE.32 #MinInt+1,{W}-3*4 alt start (store flag to show enabling is occuring)	
2 4 F 4	F/44	*altwt*	MOVE.32 #-1,{W} alt wait (process is descheduled unless {W}-3*4=MinInt+3)	[AØ,BØ,CØ,EØ]
2 4 F 5	F/45	*altend*	MOVE.32 {I}+{{W}},I alt end (set I to first instruction of branch selected)	[AØ,BØ,CØ]
2 4 F 9	F/49	*enbs*	MOVE.32,AEQ #MinInt+3,{W}-3*4 enable skip (If A=1, MOVE.32 MinInt+3,{W}-3*4)	[AØ]
2 3 F 0	F/30	*diss*	MOVE.32,BEQ A,{W} disable skip (if B=1 AND the first ready process, execute the move and A:=1, A:=0 otherwise)	[A,B:=C,CØ]
2 4 F 8	F/48	*enbc*	MOVE.32,AEQ #MinInt+3,{W}-3*4 enable channel (if A = 1 AND another process using channel, execute move, A:=0 otherwise)	[AØ,BØ,CØ]
2 2 F F	F/2F	*disc*	MOVE.32,BEQ A,{W} disable channel (if B=1 AND the first ready process, execute the move and A:=1, A:=0 otherwise)	[A,BØ,CØ]

Figure 17. Alternative-construct instructions.

T414 instruction set

2 2 F 2	F/22	*ldtimer*	MOVE.32 TIMER,A	[A,B:=A,C:=B]
		load timer (A = value of current priority level clock)		
2 2 F B	F/2B	*tin*	COMP.32 TIMER,A	[AØ,BØ,CØ,EØ,HØ]
		timer input (continue execution only when ClockRegi>A)		
2 4 F E	F/4E	*talt*	MOVE.32 #MinInt+1,{W}-3*4 MOVE.32 #MinInt+2,{W}-4*4	
		timer alt start ({W}-3 = state, {W}-4 = tlink)		
2 5 F 1	F/51	*taltwt*	MOVE.32 #-1,{W}	[AØ,BØ,CØ,EØ,HØ]
		timer alt wait (process descheduled unless state = MinInt+3, or tlink = MinInt+1)		
2 4 F 7	F/47	*enbt*	MOVE.32,AEQ #MinInt+3,{W}-3*4 MOVE.32,AEQ #MinInt+1,{W}-4*4 MOVE.32,AEQ B,{W}-5*4	[A,B:=C,CØ]
		enable timer (if A=1 AND time AFTER B then newtime = B, otherwise newtime = time)		
2 2 F E	F/2E	*dist*	MOVE.32,BEQ A,{W}	[A,BØ,CØ]
		disable timer (if B=1 AND time later than guard AND first ready process, execute the move and A:=1, A:=0 otherwise)		
2 5 F 4	F/54	*sttimer*	MOVE.32 A,ClockReg0 MOVE.32 A,ClockReg1 Start clocks	[A:=B,B:=C,CØ]
		store timer		

Figure 18. Timer instructions.

2 1 F 8	F/18	*sthf*	MOVE.32 A,FPtrReg0	[A:=B,B:=C,CØ]
		store high priority front pointer		
2 1 F C	F/1C	*stlf*	MOVE.32 A,FPtrReg1	[A:=B,B:=C,CØ]
		store low priority front pointer		
2 5 F 0	F/50	*sthb*	MOVE.32 A,BPtrReg0	[A:=B,B:=C,CØ]
		store high priority back pointer		
2 1 F 7	F/17	*stlb*	MOVE.32 A,BPtrReg1	[A:=B,B:=C,CØ]
		store low priority back pointer		
2 3 F E	F/3E	*saveh*	MOVE.32 FPtrReg0,{A} MOVE.32 BPtrReg0,{A}+4	[A:=B,B:=C,CØ]
		save high priority queue registers		
2 3 F D	F/3D	*savel*	MOVE.32 FPtrReg1,{A} MOVE.32 BPtrReg1,{A}+4	[A:=B,B:=C,CØ]
		save low priority queue registers		

Figure 19. List-pointer instructions.

Timers. The transputer has two 32-bit, free-running clocks that give it a measure of time. Timer Alternate Start, Timer Alternate Wait, Enable Timer, and Disable Timer instructions (*talt, taltwt, enbt,* and *dist*) are used with alternative instructions when time is introduced into one or more of the guards (Figure 18). The uses and effects of these instructions are similar to those described in the section on process manipulation. The Load Timer instruction (*ldtimer*) reads the value of the current priority clock and places its value onto the evaluation stack. Time delays are possible by using the Timer Input instruction. The A register is set to a certain time. If the time in A has passed, (Inmos uses the notation ClockReg AFTER A), the instruction has no effect. If it is in the future (A AFTER ClockReg or A = ClockReg), the process is removed from the schedule. Unsigned modulo arithmetic operations must be used to prevent errors when the clock wraps around from the

most positive to the most negative value. The Store Timer instruction (*sttimer*) stores the value from register A into both clock registers (ClockReg0 and ClockReg1) and then starts both clocks.

Pointers. The registers for the construction of linked lists appeared in Figure 2 (FPtrReg0, FPtrReg1, BPtrReg0, and BPtrReg1). The following instructions manipulate these registers (Figure 19). A set of instructions load these registers: *sthf, stlf, sthb,* and *stlb.* These instructions store(s) the value in register A into the appropriate register (*h* for high priority, *l* low, *f* for front pointer, *b* for back). The Save High- and Low-Priority Queue Registers instructions (*saveh* and *savel*) save the front and back pointers in two consecutive locations, the first pointed to by register A. Again, *h* is high priority, and *l* is low priority.

This article has attempted to illustrate the internal workings of the T414 transputer using standard microprocessor notations. We provide more details and program examples elsewhere to explain the last four groups of instructions, which are rather tricky.[7]

The transputer is not an ordinary microprocessor. It takes some concepts from RISCs and also uses a stack for evaluation of arithmetic and logical operations rather than registers. Most importantly, it has been designed to work in a concurrent environment. Consequently, the transputer has a number of complex and cumbersome instructions not normally found in other microprocessors. However, these instructions can be rather clearly expressed by a set of familiar instructions.

References

1. M. Homewood et al., "The IMS T800 Transputer," *IEEE Micro*, Vol. 7, No. 5, Oct. 1987, pp. 10-26.
2. *IMS T414 Transputer Reference Manual*, Inmos Ltd., Bristol, England, 1984.
3. J.D. Nicoud and F. Wagner, *Major Microprocessors: A Unified Approach Using CALM*, North Holland Press, Amsterdam, 1987.
4. J.D. Nicoud and P. Fah, *CALM Standard and Explanations*, 2nd ed., Laboratoire de Microinformatique, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, Dec. 1986.
5. *The Transputer Instruction Set—A Compiler Writer's Guide*, Inmos Ltd., Bristol, England, 1987.
6. C. Plumb, "An Introduction to Transputer Assembly Language," Electronic paper broadcast on UUCPnet/Bitnet from ccplumb@watmath.waterloo.edu, Jan. 1989.
7. A.M. Tyrrell and J.D. Nicoud, "Scheduling and Parallel Operations on the Transputer," to be published in *Proc. Euromicro Conf.*, North Holland Press, 1989.

Jean-Daniel Nicoud is a professor at the Ecole Polytechnique Federale de Lausanne in Switzerland. He has been active in microprocessor-related research for 16 years and has designed many microprocessor-based systems. His interests and those of his research group include microcomputer design, development tools, local networks, microcomputer peripherals, neural networks, and very large scale integration technology.

Nicoud holds a degree in engineering physics and a PhD degree in electrical engineering, both from the Ecole Polytechnique Federale de Lausanne. He was an associate editor of *IEEE Micro* from its inception in 1981 until 1985. He is a member of the IEEE Computer Society.

Andrew Martin Tyrrell is a senior lecturer in the Department of Electrical, Electronic, and Systems Engineering at Coventry Polytechnic, Coventry, England. While on leave in 1988, he worked as a research fellow with J.D. Nicoud on multiprocessor systems at EPFL. His research interests are fault-tolerant software, distributed processor systems, benchmarking for multiprocessor systems, models for concurrent systems, and topologies for image-processing applications.

Tyrrell received a degree in electronic engineering from Bolton Institute of Technology, Bolton, England. He performed graduate research at Aston University, Birmingham, England, in a program for the design of fault-tolerant, loosely coupled, distributed systems. He is a member of the IEE and the IEEE.

Questions about this article can be directed to J.D. Nicoud at Ecole Polytechnique Federale de Lausanne, Laboratoire de Microinformatique (EPFL-LAMI), Av. de Cour, 37, CH-1007 Lausanne, Switzerland.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 162 **Medium** 163 **High** 164

A Logical Design Tool for Relational Databases

Here's a way to prevent anomalies from affecting the design of your relational database.

M. Mehdi Owrang O.
W. Gamini Gunaratna

The American University

New trends in software development point to the automation of various phases of the software life cycle,[1] particularly the design phase. A typical example is computer-aided software engineering technology,[2-4] which provides tools to assist in software development. In the past few years, interest has grown in the development of database-management systems (DBMSs) based on the relational model[5-7] (see the accompanying box for definitions and notations).

Systems for managing large-scale databases under the relational model (for example, Oracle[8]) have become commercially available, which makes the value of a design tool for the relational model obvious. This fact motivated us to implement the Logical Design Tool (LDT), which can also be used as an educational tool for the relational database. Using this tool during database design can detect certain problems that arise when the data is manipulated. An example is a cycle in a schema[6,7,9] that leads to ambiguous interpretation of queries. The major problem, however, is to come up with a particular normal-form design (such as a third normal form) through the process of decomposition. Decomposing a relation scheme into several relation schemes prevents the occurrence of anomalies.[5-7] Consider the following Suppliers relation scheme taken from Ullman.[7]

SUPPLIERS (SNAME, SADDRESS, ITEM, PRICE)

We can see several problems with this scheme:

- **Redundancy.** The address of the supplier repeats each time an item is supplied.
- **Potential inconsistency (update anomalies).** Because of the redundancy, users can update the address for a supplier in one tuple and leave it fixed in another. Thus, users may not have a unique address for each supplier as they feel intuitively that they should.
- **Insertion anomalies.** Users cannot record an address for a supplier if that supplier does not currently provide at least one item. Null values can be used for the Item and Price components of a tuple for that supplier. However, when users enter an item for that supplier, they may not remember to delete the tuple with the null values. In addition, if Item and Sname form a key for the relation, it might be awkward or impossible to look up tuples with null values in the key.
- **Deletion anomalies.** If users delete all the items supplied by one supplier, they unintentionally lose track of its address.

0272-1732/89/0600-0076$01.00 © 1989 IEEE

The Relational Database Model

Here we present some brief definitions and notations of the relational database used in this article.

Definitions

Conceptually, a table can represent a *relation*. Each row represents a *tuple*, and each column represents an *attribute*. The set of attribute names for a relation is called the *relation scheme*.[5-7,10] In Table A, we see a relation whose attributes are S#, Sname, Status, and City. The tuple is {S1, Smith, 20, London}. The relation scheme for this relation is SUPPLIERS (S#, Sname, Status, City). Any set of one or more attributes that uniquely identifies the tuples is called a *candidate key* (like S#), and the attribute selected for the relation is called the *primary key* (or just *key*). A *relational schema* is composed of a set of relation schemes.[5-7,10]

Table A.
The Suppliers relation.

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Data dependencies

Data dependencies are the semantic constraints imposed on data. Recognizing the different types of dependencies is part of the process of understanding what the data means.[5-7,10]

Functional dependency (FD). Given a relation R, attribute Y of R is functionally dependent on attribute X, denoted as $X \rightarrow Y$, if and only if each X value in R has associated with it precisely one Y value in R. For example, attributes Sname, Status, and City of relation SUPPLIERS in Table A are each functionally dependent on attribute S#, because, given a particular value for S#, one corresponding value exists precisely for each of the attributes Sname, Status, and City in symbols:

S# $\rightarrow$ Sname, S# $\rightarrow$ Status, S# $\rightarrow$ City

Multivalued dependency (MVD). A multivalued dependency, denoted as $X \twoheadrightarrow Y$, means that the X value determines a set of Y values.

Join dependency (JD). Functional and multivalued dependencies together capture a significant amount of semantic information of the data. For example, the FD $X \rightarrow Y$ implies that the database scheme $\{XY, XZ\}$ is information that is equivalent to the relation scheme XYZ. This information can be represented concisely by the join dependency JD $\bowtie [XY, XZ]$. The set of join dependencies associated with a scheme R determines exactly those database schemes that can represent R without losing information.

Normal forms

In the theory of database design,[5-7,10] attributes can be assembled into a set of relations in many different ways. Some arrangements are "better" than others. The ways in which attributes can be arranged are called normal forms (that is, third normal form (3NF), Boyce-Codd normal form (BCNF), and fourth normal form (4NF)).

The theory behind the arrangement of attributes into relations is known as *schema normalization*. Normalization allows us to recognize certain undesirable properties of relations (such as data redundancy or inconsistency) and shows how such relations can be converted to a more desirable form.

Third normal form. A relation R is in 3NF if and only if the nonkey attributes (attributes that do not participate in the key of the relation), if any, are

- mutually independent (two or more attributes are mutually independent if none of the attributes concerned is functionally dependent on any of the others); and
- fully dependent on the key of R (attribute Y of relation R is fully functionally dependent on attribute X of relation R if it is functionally dependent on X and not functionally dependent on any proper subset of X).

Boyce-Codd normal form. A relation R is in BCNF if and only if every determinant (any attribute on which some other attribute is fully functionally dependent) is a candidate key.

Fourth normal form. A relation R is in 4NF if and only if—whenever an MVD exists in R, say $A \twoheadrightarrow B$—all attributes of R are also functionally dependent on A. In other words, the only dependencies (FDs or MVDs) in R are of the form $K \to X$ (for example, a functional dependency from a candidate key K to some other attribute X).

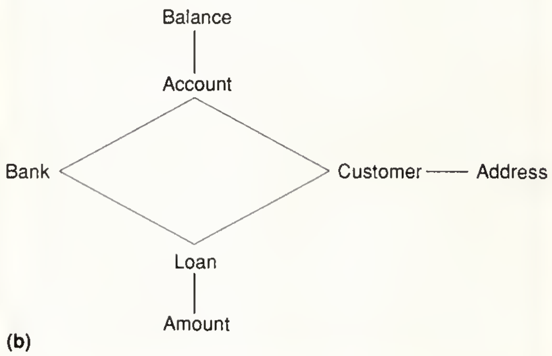
Decomposition

Relations are transformed from one normal form to another by decomposing them into a set of smaller relations.[5-7,10] The decomposition of a relation scheme $R = \{A_1, A_2, \ldots, A_n\}$ is its replacement by a collection $P = \{R_1, R_2, \ldots, R_k\}$ of subsets of R such that $R = R_1 \cup R_2 \cup, \ldots, \cup R_k$. Such decomposition should be lossless, which means that if we join the relations r_i ($i = 1, 2, \ldots, k$) for R_i ($i = 1, 2, \ldots, k$), we obtain the original relation r for R (no information is lost).

Database acyclicity

We would generally like database schemes to be acyclic.[6,7,9] One reason is that a unique connection between attributes is desirable. Cyclic schema causes some problems in interpreting queries because multiple paths connect a set of attributes involved in the queries. For example, consider the banking example of Figure A that represents a cyclic schema. If a query requests those banks associated with a given customer, it is not clear whether the desired information is the set of banks where the customer has loans, or the set of banks where the customer has accounts, or both.

R1 (Account, Balance)
R2 (Account, Bank)
R3 (Bank, Loan)
R4 (Loan, Amount)
R5 (Loan, Customer)
R6 (Customer, Address)
(a) R7 (Account, Customer)



(b)

Figure A. Examples of a relational schema (a) and a corresponding graph containing a cycle (b).

This example becomes nonproblematic if we decompose the relation scheme into the following two relation schemes.

SA (SNAME, SADDRESS)
SIP (SNAME, ITEM, PRICE)

Note that none of the commercial relational database-management systems (RDBMSs) have any facilities to assist users in designing a logical view of data to prevent these anomalies from being built into the actual database. A design tool such as the LDT can provide the necessary assistance to users so that they can process the relation schemes and decompose them when it is required.

The LDT is mainly based on the concept of *schema normalization*.[5-7,10] We have implemented different operations in the LDT to assist the designer of the relational database to manipulate relation schemes to see whether they are in certain normal forms (third, Boyce-Codd, or fourth[5-7,10]). If not, these operations decompose the relation schemes into certain normal forms if so desired. In addition, the LDT includes other operations that test for both the acyclicity of relation schemes[6,7,9] and what we call "lossless" join decomposition.[5-7,10] These operations enable the designer to detect other undesirable problems such as a cycle in schema that causes ambiguity in the interpretation of some queries.

Overview of the LDT

We wrote the program in Turbo Pascal for the IBM PC AT/XT so that users can easily expand or modify it. This window-based, menu-driven package contains a help file that can be accessed at any level of the program. Users can define both the relation schemes and the dependencies.

The LDT contains a built-in editor to select the operations. Users can input data either from the keyboard or an input file. They can also add or delete data to and from the schema or dependency tables. The LDT has a maximum static limit for the attributes in a relation or a dependency. However, removing these restrictions does not disturb the structure of the program in any way.

Here we briefly describe the structure and syntax of the relations and dependencies used in the LDT.

Data structure. The first structure is the symbol table, SYM_TAB. The symbol recognizer stores each attribute name that it identifies in this table as shown in Figure 1. It then checks attributes contained in dependencies against the symbol table to ensure the validity of the dependencies.

Figure 2 demonstrates the relation schemes table, SCH_TAB. Each relation scheme identified by the sentence recognizer is stored in this table.

Figure 3 shows the data dependencies table, DEP_TAB. Each dependency identified by the sentence recognizer is stored in this table.

External files. The LDT checks two files— schema (ssch) and dependency (ddep)—when users first employ the system. If the files are not empty, the data in ssch and ddep is restored to tables SCH_TAB and DEP_TAB, respectively. When users quit the system, it backs up the data in tables SCH_TAB and DEP_TAB to the same files for the next use.

The syntax for relations and dependencies is as follows, beginning with relations:

<legal_id> (<legal_id> [,<legal_id>])
example: bank(account, customer_name, customer_add)

The syntax for a functional dependency is

FD <legal_id> [,<legal_id>] | <legal_id> [,<legal_id>]
example: FD account | customer_name, customer_address

The syntax for a multivalued dependency is

MVD <legal_id> [,<legal_id>] | <legal_id> [,<legal_id>]
example: MVD supplier_num | supplier_name, location

A join dependency is as follows:

JD <legal_id> [,<legal_id>] |...| <legal_id>[,<legal_id>]
example: JD a, b, c | c, d, e | e, f, g

SYM_TAB [i] ::=

Attribute name	
Next	Code

Code	Integer (>=0), which represents the attribute name
Next	Pointer (>=0) to the next component

Figure 1. Symbol table.

SCH_TAB [i] ::=

Count
Relation name
Attributes

Attributes	Integer codes that represent attribute names
Count	Sequence number of the relation scheme

Figure 2. Relation schemes table.

DEP_TAB [i] ::=

Count	Dep-type
Attributes	

Attributes	Integer codes that represent attribute names
Count	Sequence number of the data dependency
Dep-type	Type of data dependency

Figure 3. Data dependencies table.

Functions of the LDT

Here we describe functions that can assist the designer to develop a logical view of the relational database.[5-7,10] In the relational model, data is organized into a set of relation schemes and data dependencies that describe the interrelationships between the data. As stated, in the logical design of a relational database, normalizing the relation schemes can prevent anomalies that occur at the time of manipulation of data. Normalization is a process of decomposing a relation scheme into several relation schemes based on the existing data dependencies. Consequently, we have considered a major issue in the implementation of the

LDT, namely, the schema-normalization problem. We have considered three kinds of dependencies (functional, multivalued, and join) and three normal forms (third, Boyce-Codd, and fourth).

We summarize the functions of the LDT as follows. The system

• detects whether or not a data dependency can be inferred from a given set of data dependencies;
• detects whether or not a given set of relation schemes and data dependencies is in the third, Boyce-Codd, or fourth normal forms;
• decomposes a relation scheme into either Boyce-Codd or fourth normal forms;
• detects whether or not a given set of relation schemes has a lossless-join property corresponding to a set of data dependencies; and
• detects whether a given set of relation schemes is acyclic.

Figure 4 depicts the overall LDT system diagram. At this point, the LDT has 14 commands; four of them construct the I/O module. The I/O module is responsible for processing the relation scheme and data dependencies and creating the schema and dependency tables. The heart of the LDT is the schema-normalization module that enables users to process these tables in constructing the relational schema in the desired normal form.

To use the normal forms and decomposition functions, users have the option either to employ the relation schemes and dependencies stored in the ssch and ddep files or to provide new sets of them. Note that the schema-normalization module uses some other functions such as key finding and chase that are not available to users. The key-finding[6,7] function processes the attributes of a relation scheme with the associated dependencies to determine a set of candidate keys for the scheme. The chase[6,7] function processes the dependencies, attributes, and relation schemes represented as a table. It then produces a new table, with respect to the dependencies, that can be used by the key-finding and lossless-join functions.

In the other module, we have included the following functions to improve the capability of the LDT.

First, a dependency-inference function enables users to eliminate any dependency that is implied by the

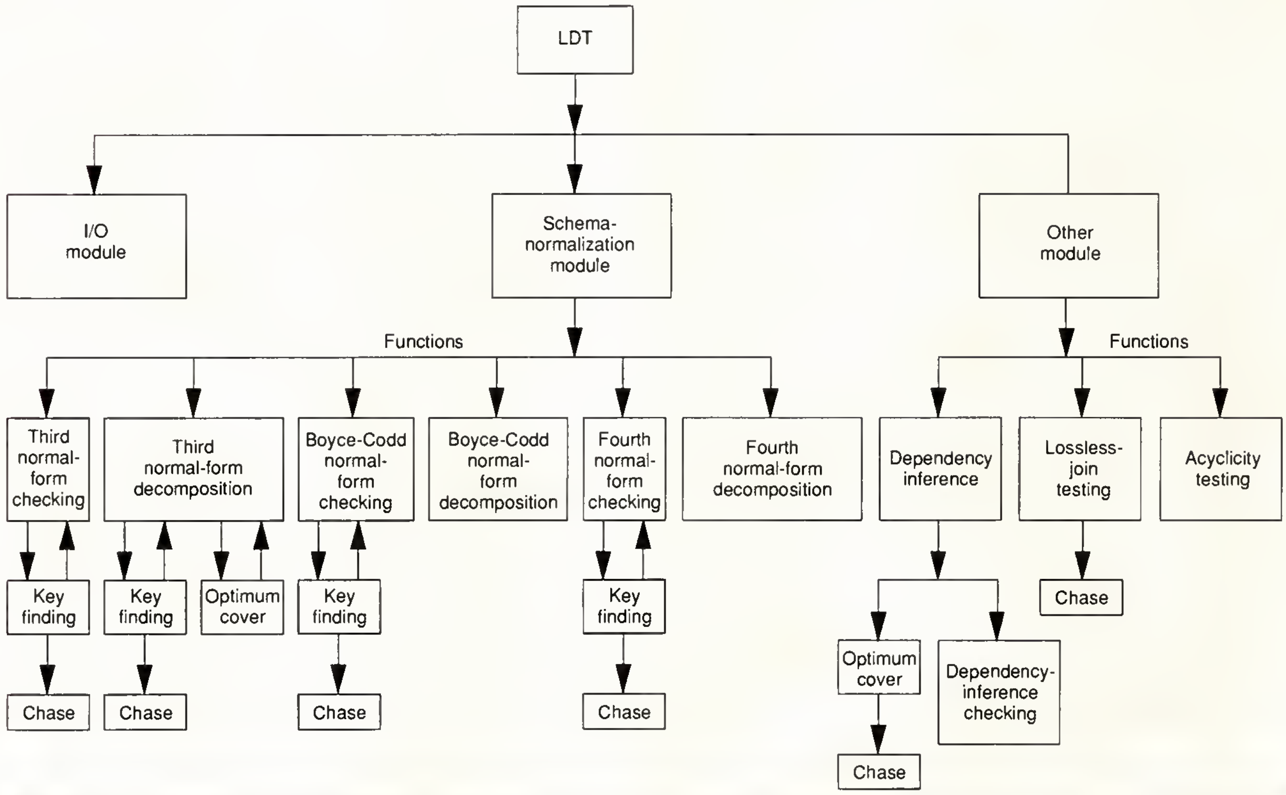


Figure 4. The overall LDT system.

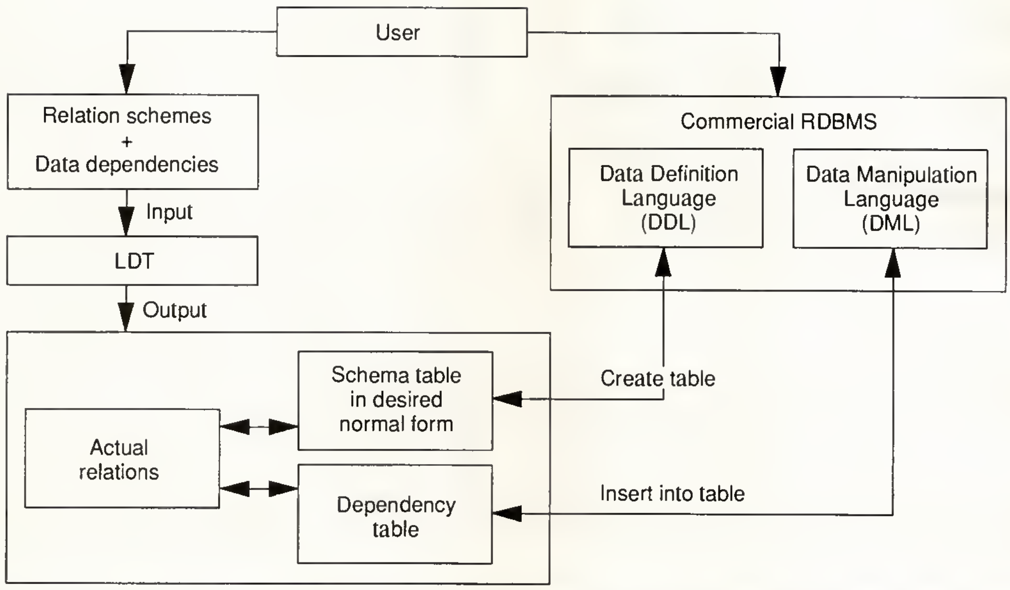


Figure 5. Connection between the LDT and the RDBMS.

existing dependency. This function also uses the optimum-cover function to define a minimum set (FD_s) that have no redundancy. The dependency-inference function is very useful when the dependencies are input, because users add to the dependency table only those dependencies that are not implied by existing ones.

Second, the lossless-join function enables users to see whether the decomposition of a relation scheme to several relation schemes has the lossless-join property. This property guarantees that the original relation will not be changed as a result of the decomposition.

Finally, the acyclicity-testing function enables users to test whether a set of relation schemes is acyclic. As mentioned, the cyclic schema is not desirable as it introduces ambiguity in interpretation of the queries. Therefore, it would be very beneficial to test for schema acyclicity before the actual database is built.

In general, database schemes should be acyclic.

Application

As mentioned, the LDT enables the user to manipulate the data at the logical level by deriving the desired logical view of the data. The output generated by the LDT can feed directly into a commercial relational database software package like Oracle given the appropriate interface between the LDT and the RDBMS. Figure 5 illustrates this connection.

During normal RDBMS operation, users employ the create-table operation to generate the relation schemes. They then use the insert-into-table operation to add tuples into the relations. In most cases, users do not know whether these schemes are in a particular normal form or whether these schemes construct an acyclic schema. As a result, users may build a database with a potential for anomalies.

In the new environment in which the RDBMS utilizes the LDT, users provide the relation schemes and dependencies to the LDT. Then they use the system to put the relational schema in the desired normal form. Upon exiting from the LDT, the relation schemes in the desired normal form and the dependencies are saved in the schema and dependency files, respectively. The RDBMS can use the schema file to create the relation schemes. Users can employ the insert-into-tables operation to store the actual tuple in the relations. However, the RDBMS now can use the dependency table to satisfy the dependencies (RDBMSs do not check for validity of the tuples to be inserted with respect to the dependencies).

The LDT can be implemented on mainframes without any major modifications.

It is obvious that enhancing the RDBMS with the LDT restricts users in terms of arbitrarily creating and deleting the relations. However, this new configuration (Figure 5) improves the data integrity within the database because most of the data anomalies that might occur at the logical level can be eliminated by using the LDT before building the actual database using the RDBMS.

It is worthwhile mentioning that the design of the interface between the LDT and the RDBMS in Figure 5 can easily be applied to any other commercial RDBMSs (like IBM's DB2 and SQL/DS, and Cincon Corporation's Supra[5]) provided that the LDT functions are implemented in the operational environment of the RDBMS in use. For instance, the LDT functions can be implemented using Standard Pascal on an IBM mainframe, which can then link to the DB2 relational database system. In general, LDT functions can easily be incorporated into a host language (like IBM's PL/I) as a set of calls in the same way as the Data Manipulation Language (DML) statements are incorporated when the RDBMS package is used as a batch environment. In addition, LDT functions can be used independently prior to the use of the RDBMS package in an interactive environment as described in this section (note that DB2 supports both batch and interactive environments).

The process described in this section can provide the basis for developing a common interface between the LDT and any RDBMS operating on any computer for the following reasons:

- All commercial RDBMS packages operate on relations.
- The structures used in the LDT implementation are simple and do not require any special feature that might not be supported by some computers.
- New trends in software development make it possible to develop a portable software[11] using such methods as microcoding.

Expandability and hardware

The LDT has been implemented on the IBM PC AT/XT with a minimum memory of 256 Kbytes. We added a number of small, device-dependent routines to make this program easy to use. Device- and compiler-dependent functions can be modified without changing the basic functions of the LDT. Hence, the LDT can be implemented on mainframes without making any major modifications.

Examples of functions

We tried the following examples on the LDT system. (Underlined attributes denote the key for the relation.)

Normal forms checking. Given the relation scheme

R (<u>CITY, STATE</u>, ZIP_CODE) with dependencies
fd CITY, STATE | ZIP_CODE
fd ZIP_CODE | CITY

the LDT responds

: relation R is in third normal form;
: relation R is not in Boyce-Codd normal form.

Lossless-join testing. Given decomposition of the previous relation scheme *R* into relation schemes

R1 (STATE, ZIP_CODE)
R2 (CITY, ZIP_CODE)

the LDT responds

: This is a lossless join.

Decomposition. Given the following relation scheme and dependencies,

DWELLER (<u>NAME</u>, ADDRESS, APT#, RENT)
fd NAME | ADDRESS, APT#
fd ADDRESS, APT# | RENT

the LDT responds

: The relation scheme Dweller is not in third normal form.

Then the LDT decomposes that scheme to the third normal form as follows.

TENANT (<u>NAME</u>, ADDRESS, APT#)
APARTMENT (<u>ADDRESS, APT#</u>, RENT)

Schema-acyclicity testing. In terms of the relational schema for a banking database given earlier in Figure A, the LDT generates a message to indicate that the schema is cyclic. The attributes Account, Bank, Loan, and Customer compose the cycle. Figure A also represents the graph corresponding to the banking system.

We have presented a logical design tool to assist designers of logical relational databases. The LDT enables the designer to manipulate the data at the logical level to derive the desired view of the data. The output generated by the LDT can feed directly into commercial relational database software packages.

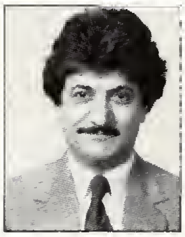
We are currently expanding the LDT to include capabilities for automatic generation of functional and multivalued dependencies and automatic conversion of a cyclic relational schema to an acyclic schema. Another intriguing direction is enhancing the user interface to make the LDT more of an expert system that can understand the designer's context and goals and give useful advice about alternative design choices and their implications.

Acknowledgments

We thank the reviewers for their suggestions, which have resulted in several improvements over our original manuscript.

References

1. S.L. Pfleeger, *Software Engineering—The Production of Quality Software*, MacMillan Co., Inc., New York, 1987.
2. A.F. Case, Jr., *Information Systems Development Principles of Computer Aided Software Engineering*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
3. *CASE Quarterly*, Vol. 1, No. 1, Apr. 1987.
4. J. Martine and C. McClure, *Structured Techniques—The Basis for CASE*, Prentice-Hall, 1986.
5. C.J. Date, *An Introduction to Database Systems*, Vol. 1, 4th ed., Addison-Wesley Pub. Co., Reading, Mass., 1986.
6. D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Md., 1983.
7. J.D. Ullman, *Principles of Database Systems*, Computer Science Press, 2nd ed., 1982.
8. *ORACLE*, Oracle Corp., Belmont, Calif.
9. C. Beeri et al., "On the Desirability of Acyclic Database Schemes," *J. ACM*, Vol. 30, No. 3, July 1983, pp. 479-513.
10. E.F. Codd, "Further Normalization of the Database Relational Model," in *Current Computer Science Symposia*, Vol. 6, *Data Base Systems*, Prentice-Hall, 1971, pp. 33-64.
11. P.J.L. Wallis, *Portable Programming*, The Macmillan Co., London, 1982.



M. Mehdi Owrang O. is an assistant professor of Computer Science and Information Systems at the American University, Washington, D.C. His research interests include software engineering, databases, and query translation in the distributed database environment.

Owrang received the MS and PhD degrees in computer science from the University of Oklahoma at Norman. He is a member of the Association for Computing Machinery, the IEEE, the Sigma Xi (American Scientist Society), and the IEEE Computer Society.

W. Gamini Gunaratna is currently a research associate for the Center for Bio Analytical Research (CBAR) at the University of Kansas. He was a graduate student in computer science at the American University when he coauthored this article. He has been a guest scientist at the National Bureau of Standards, Scientific Computing Division. His research interests are mainly in the area of CASE studies and database and algorithm optimization techniques.

Gunaratna received the BSc degree from the University of Colombo, Sri Lanka, and the MS degree in computer science from the American University, Washington, D.C.

Questions regarding this article may be directed to Mehdi Owrang, the American University, Department of Computer Science and Information Systems, 4400 Massachusetts Ave., N.W., Washington, D.C. 20016.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 165 **Medium** 166 **High** 167

Department

Micro Law

Richard H. Stern
Law Offices of Richard H. Stern
1300 19th Street NW, Suite 300
Washington, DC 20036

Appropriate and inappropriate legal protection of user interfaces and screen displays, Part I

Whether and how user interfaces and screen displays for computer programs should be protected against competitive imitation is an area currently generating considerable legal controversy. Proponents of such protection assert that proprietors of these highly valuable adjuncts of computer software need and deserve protection against copying. Others question whether such protection might more hinder than promote software progress.

Those who conclude or assume that there should be legal protection for user interfaces and screen displays disagree over the preferred legal mechanism. Those who conclude or assume that the mechanism should be the present federal copyright laws disagree over the method of protection. Should user interfaces and screen displays be placed under a general umbrella of legal protection extended to the computer programs to which they relate? Or should they instead be protected separately as things like pictures? Of major concern to software entrepreneurs and software users is which aspects of user interfaces and screen displays should be reserved in the public domain, which should be legally protected, and how far the protection should go.

The conflicting opinions and general uncertainty led the US Copyright Office, the federal agency charged with administering the copyright system, to solicit industry comments. The Office held a public hearing on these issues in September 1987 to aid it in setting its policy. The IEEE Computer Society's Board of Governors passed a resolution favoring protection of screen displays (see adjacent box) and transmitted it to the Copyright Office.

Resolution

Resolved, that the Board of Governors of the Computer Society of the IEEE, in order to encourage and promote creativity and investment in this aspect of the computer graphics field, recommends that the Copyright Office of the Library of Congress should permit an owner of a computer program screen display to register the screen for copyright protection apart from the registration of the underlying code in the computer program that generates the display.

A panel of witnesses from the Computer Society testified before the Copyright Office, presenting a spectrum of views. Although different Computer Society panelists had different degrees of enthusiasm for a regime of legal protection of screen displays and user interfaces, their consensus was that some protection was desirable. They also agreed that the kind of protection should be independent of that for whatever computer code or computer program was used to generate screen displays.

The three IEEE Computer Society witnesses were S. Levine, A. Peller, and myself. Levine expressed apprehension lest excessive legal protection stifle technological progress. Peller was convinced that the incentives of protection would contribute far more socially than they could hinder progress. I felt protection would probably promote progress more than hinder it if the Copyright Office was very careful in how it administered the law.

Despite the possible differences in approach taken by the Computer Society witnesses, it is fair to say that all of them at least tacitly agreed with the premise that progress in software and software innovation is to be desired. That premise also underlies the present analysis. Probably these witnesses all accepted, as well, the further premise that increased public acceptance of computers as a part of everyone's daily life is socially desirable, and that proliferation of usage of computers and computer software in many aspects of life ought to be encouraged as a means toward development of a better society. That agenda is also a philosophical premise of this analysis.

After a nine-month gestation period, in June 1988 the Copyright Office issued a policy statement governing copyright registrations of screen displays. Notwithstanding judicial precedent prescribing separate copyright treatment of computer programs and screen displays, the Copyright Office determined that screen displays should be regarded as an "integrally related" part of the computer programs to which they relate, rather than as separately protectable works. Accordingly, the Office now refuses to register screen displays separately as distinct works of authorship, and registers them and the computer programs whose code generates them as a single package.

Implicit in the Copyright Office's policy statement, but left undiscussed, is the Office's apparent conclusion that screen displays are or should be in some way protected to some extent by existing copyright law. Discussed only insofar as the Office stated that it refused to opine on the subject is the issue of what is the proper scope of such copyright protection in screen displays.

Some members of the Computer Society's Committee on Public Policy felt the Copyright Office's action was so wrong-minded and potentially damaging to software progress that it should be subjected to judicial review or an attempt should be made to overturn it by legislation.[1] A poll of interested members, however, led to a majority vote for taking no action at this time and instead for awaiting further developments.

This column is the first part of a series of Micro Law columns evaluating the arguments for and against the protection of user interfaces and screen displays, and possible legal mechanisms for their protection under existing or new laws. It is an expansion and restatement of parts of the written statement submitted to the Copyright Office on behalf of the Computer Society. Here and in the next issue I discuss the possible difficulties and problems that protection of user interfaces and screen displays might cause.

Subsequent columns conclude that protecting user interfaces and screen displays under the existing copyright laws is, despite some risks, on balance and although rather narrowly, the most desirable course of legal action. The Copyright Office now requires that screen displays be legally intertwined with their associated computer programs. This, I suggest, is a serious mistake. Treating them as distinct pictorial, audiovisual, or other works, as the Computer Society urged, would be better.

Definition of subject matter

The *user interface* for a computer program may be defined as the means by which users of the program interact with it to input data, to direct the performance of procedures, and to receive information produced by the program. While this term is too novel to be well defined yet, one could understand it in its most literal sense. That is, the user interface is the boundary between a user and a computer system, just as in physics an interface is the surface between any two phases of a system, such as ice and water, water and air, and air and ice—in the case of a glass of ice water. If you hold a glass of ice water, an interface exists between the outside of the glass and the outside of the skin of the palm of your hand. That terminology or usage is not helpful or idiomatic, however, in the present context, and it does not relate immediately to interesting and controversial legal questions.

The computer's keyboard is a mechanical interface; keystrokes are the user's method of interfacing.

There are other interfaces found in computer-related technology. For example, protocols for interconnection of computers and peripheral equipment are sometimes termed interfaces. Sets of commands used in computer languages or for interaction of computer programs are also sometimes termed interfaces. In this and following columns, I use the term *user interface* only to refer to interfaces between human users of computer programs and computer equipment used to execute the computer programs.

Probably the most common forms of user interface are sets of keystrokes by means of which a user inputs data to a computer or directs performance of procedures in a computer, via the keyboard of the computer. Visual patterns appear on the screen of the cathode-ray tube monitor for the computer (screen displays), by which the user perceives outputs, receives cues for making inputs, and receives other information from the computer and computer program. Those, too, are user interfaces.

It may be helpful in some contexts to distinguish between a) the mechanical or physical aspect of a user interface, that is, a device, and b) the aspect that is the use or method of using an interface device. Thus, the computer's keyboard is a mechanical interface device, while the keystrokes are the user's method of interfacing by means of a keyboard (that is, the user presses the keys of the keyboard and causes keystrokes). The former is tangible and relatively permanent; the latter is intangible and ephemeral.

User interface devices of possibly increasing future importance to microcomputer systems are touch screens and voice-actuated devices. These permit users to input commands by touching a part of the screen and by speaking into a microphone to the computer. (By the same token, the computer can "talk back" via a loudspeaker.)

Special-purpose computer systems may utilize other user interface means. For example, a coin-operated video game machine is a special-purpose microcomputer in which the user (game player) inputs signals (data) to the computer by moving a joystick or trackball and pressing buttons. Still another interface device permits players to move their hands, thereby changing capacitance or some aspect of a field, and the motions are translated into input signals. A user performs these acts in response to another aspect of the user interface for the video game, the screen display.

Such user interfaces for computer systems find their parallels in other types of electrical and mechanical systems. For example, the steering wheel and accelerator pedal are user interface devices for operating automobiles. The steering wheel acts both as an input interface device and as an output interface device, since the driver uses the steering wheel to cause the car to change direction and at the same time gets feedback from its varying resistance, which gives the driver a "feel" for road conditions. A gearshift is another automobile interface device, as is a brake pedal.

In general, any physical system of which a human being is a part or with which a human being interacts must have a user interface.

In many systems, the user interface is not something of great economic value or, at least, not much legal controversy. User interfaces for computer programs, however, are of great economic value and are becoming the subject of increasing legal controversy. Several recent court decisions and pending lawsuits involve assertions of rights under the copyright laws to particular user interfaces for computer programs. As a result, considerable industry concern has been expressed over the implications of these assertions of proprietary rights.

This series explores the kind of legal rights that might be asserted over *nondevice* aspects of user interfaces, of which

Hardware Considerations

Hardware considerations appropriate for this discussion include displays and methods of input.

Displays

The ordinary form of screen display for most present microcomputer systems is a monochrome or color cathode-ray tube monitor, usually a raster-scan television monitor. Other display technologies now exist, however, and are coming into increasing use. These include plasma, liquid crystal, and electroluminescence. The design choices available to screen designers differ with these different display technologies, and software must be written differently to take advantage of the unique characteristics of the various display means. The legal and policy points discussed in this series, however, do not turn on the particular display technology that is used.

Input

The ordinary method for a user to input commands or other information to a computer is pressing keys on a keyboard connected to the computer (entering keystrokes). For example, in using a computer program associated with a menu, a user may press an alphanumeric key or keys and then the Return or Enter key to command the computer to perform a function. An example might be entering "P" to print a document. In the remainder of this series, I use the convention of representing keystrokes by angle brackets (for example, <P> for keystroke "P").

Other input user interface devices now in use include the joystick, trackball, mouse, touch screen, and microphone. As in the case of display technologies, different input technologies have different unique characteristics and require different software, but they do not appear to raise significantly different legal or policy issues.

menu screen displays are illustrative. Devices, such as joysticks and touch screens, are usually protected by patents or other well-understood legal systems, and they ordinarily do not raise the perplexing issues with which this series will struggle. (Hardware considerations relevant to this discussion appear in the accompanying box.) The series principally discusses legal rights in screen displays for computer programs, and secondarily discusses legal rights in sets of keystrokes and in other possible aspects of user interfaces that are generalizations of, or extrapolations from, keystrokes.

The subject matter involves more than the enormous amounts of money that are at stake, as between the parties. But that cannot be ignored. The installed base of Lotus's 1-2-3 spreadsheet programs, for example, has been estimated at more than 3 million copies. Current retail prices are several hundred dollars per copy.[2] Lotus's 1987 revenue from 1-2-3 was approximately $260 million.[3] To protect this market from competitive encroachments, Lotus has brought copyright infringement actions against two marketers of 1-2-3 "workalikes." The complaints allege that the defendants copied the 1-2-3 user interface to divert customers from the more expensive Lotus program.[4,5]

In addition to the money at stake, there are significant public interest implications of decisions to protect and refuse to protect user interfaces. Moreover, the questions raised in the debate over protecting user interfaces are those raised continually, whenever protection is sought for some new aspect of software technology. The present controversy thus paradigmatically brings forward the arguments and problems that will be debated for years to come. Such controversy appears each time a new form of software technology emerges and would-be proprietors of rights in such technology seek legal insulation from competitive imitation.

Intrinsic and habit aspects

Those who market software care a great deal about ownership of user interface rights, because the user interface of a computer program is a major factor in determining its commercial success.[6] There are two major aspects of the relation between a user interface and a computer program's commercial success. They may be termed the intrinsic aspect and the habit aspect, respectively.

The intrinsic aspect or qualities of a user interface concern its user friendliness. That concept requires considerable elaboration. But for the moment user friendliness may be equated with the ease with which users can learn how to use a computer program and the ease with which they can then perform tasks with it. Some user interfaces make for a user-friendly computer program, while others exasperate users. A high correlation, to say the least, exists between a computer program's user friendliness and its commercial success.

The habit aspect of a user interface involves the fact that users of accounting and other business computer programs display great reluctance to learn how to use a new user interface, once they have taken the trouble to learn how to use one user interface. It also costs money to train employees to use a new interface, and they make mistakes while habituating themselves to it.

Accordingly, a competitor's legal inability to utilize or appropriate (or misappropriate, depending on how you stand) an established user interface may greatly hinder the competitor from marketing a competitive program. By the same token, the prospect that clone-makers will be legally disabled from misappropriating the user interface of a program, once it is commercially successful, may facilitate the financing of a new software venture and spur software creativity. Thus, customer habituation to a user interface, apart from the intrinsic user friendliness of the interface, correlates with commercial success for the program.

To be sure, the speed and capabilities of a computer program have an effect on its commercial success. So does the quality of the documentation and the user support (hand-holding) made available to customers. So does advertising and promotion. User interface is not the only pertinent consideration. Programs with unfriendly user interfaces have been successful, and users have switched from one program to a more powerful competitor even though they had to learn a new user interface to do so. Nevertheless, it must be recognized that the user interface is a major factor in determining commercial success. It is thus no surprise that marketers of commercial software have considered user interfaces worth litigating over. Among the reported decisions involving un-

authorized competitive replication of screen displays are two cases.[7–9] (For a brief description of a now-settled copyright infringement controversy over a computer graphics interface system, see my 1986 account in *IEEE Micro*.[10])

Code and noncode aspects of screen displays

The electronic circuitry responsible for generating a screen display associated with a computer program is controlled by a part of the computer program. Typically, the creator of a new software product decides that the product will perform a particular set of tasks or functions and that it will interact with the user by means of a particular set of screen displays and other means for input/output. The code for executing the tasks (for example, word processing, spreadsheet calculation, or sorting data) will ordinarily be distinct from the code for generating the screen displays, although the two must be linked so that they co-act. It is common to determine the set of screens and pattern of flow from screen to screen (starting with the main menu) before writing any code for the computer program.

The accompanying box shows a paradigmatic, simple main menu (Figure A). This menu illustrates the screen-to-screen flow process, interrelation of different sets of code, and other issues that will arise in the discussion that follows. The main menu shown in the figure is based on a word-processor main menu first introduced many years ago.

Programs for generating screen displays can be written in different computer languages. For example, the computer code for the menu screen display shown in Figure A could be written for an IBM PC-compatible microcomputer (PC clone) in Basic, C, DOS batch program language, 8088 assembly language, and any one of many other computer languages. The screen display generated would still look the same. Moreover, within most languages there are different ways to write a program that will create the same display. So, many different programs, which do not resemble one another, can be written to produce any particular screen display, for a given display device. It is also true that minor modifications of a computer program for a screen display will cause visually significant differences in the display.

Illustrative Main Menu

The following simple word-processing main menu illustrates various issues that will come up in this and future discussions.

The computer program with which such a menu is associated will include code for generating this menu at the start of a session and at other appropriate times. The program will also contain code that does appropriate things when menu commands are invoked. For this word processor menu, that would include such things as:

- accessing and displaying a list of documents on file, when the keystroke <I> is entered;
- bringing up another menu of selections, when the keystroke <M> is entered; and
- terminating the word processing session, when the keystroke <Q> is entered.

Entry of some keystrokes will cause "prompts" to appear on the screen. For this word processor menu, if the keystroke <E> is entered, a prompt will ask the user to enter the name of the document that is to be edited. When that name is entered by means of appropriate keystrokes, code will cause hardware operations to occur: part of the computer program will cause the reading head of the disk drive to be positioned at the beginning of the chosen document, read the start of the document into the computer's random access memory, and display the beginning of the document on the screen.

Carrying out a prompt or entering a command may cause another menu of commands to appear. Thus, if the keystroke <P> is entered, a prompt will appear on the screen asking the user to enter the name of the document that is to be printed. When that is done, the computer program will bring up another menu screen so that the user can enter commands about how to format the printing of that document. And when that is done, the computer program will send signals to the printer to initiate the printing of the document.

```
Main Menu

C = Create a new document
E = Edit an existing document
P = Print a document
I = Index of documents on file
D = Delete a document from file
M = More menu selections
Q = Quit using system

Type the right letter and press
Return.
```

Figure A. Screen display of a simple word processing menu.

Most of the litigation and legal disputes concerning screen displays that have been publicized so far have involved menus. Screen displays can also be the visual result of inputting data to a program. Such an output display may be static or animated. Programs now exist that produce pictorial displays showing the result of a computer simulation of a mathematical model of a physical system, for particular parameter inputs. (See, for example, the Simscript II advertisement on the back cover of the April 1989 issue of *IEEE Micro*.) Icons, histograms, and the like appear on a screen instead of a table of numbers; this type of display is intended to facilitate comprehension of the simulation. Animated displays of this type are similar to video game displays. It is reasonable to anticipate future litigation over such screen displays.

The time and effort spent in preparing screen displays is directed to more than just writing the actual code that generates a screen display. Much of it goes into deciding what to include in the screen display, how to relate each screen to other screens used with the same program, and how to place the information on the screen in a way that will make it easy for the user to utilize the program.

All of these may be identified with making the computer program user-friendly and with human factors analysis.

Considerably more effort is usually devoted to these noncode aspects of screen display design than to the actual coding for generation of the screen display. Moreover, the coding effort for display code is often (but is not necessarily) quite routine compared to the effort that precedes the coding. Irrespective of whether the coding is routine compared to the screen designs, the fact remains that the nature of the authorship is different, and even the persons who design the screens are often different from those who write the display or working code. (For example, it is well known that Mitch Kapor, often credited with writing Lotus 1-2-3, did not write a single line of the code. Kapor designed many of the screens. Jonathan Sachs wrote almost all of the code and designed some of the screens.)

Designing effective screens for a computer program is an important creative effort, whose results significantly affect the popularity of the computer program. A computer program may be functionally very advanced, and it may have features that competitive programs lack. But it is unlikely to be commercially successful if it is hard for the public to learn to use the program. How the set of screens associated with a computer program is designed is a major determining factor of how easy it is to use the program.

Like everything else, the creation of screen displays and other aspects of computer program user interfaces, and thus progress in this aspect of computer graphics, has to pay its own way. Somebody has to pay the salary of screen designers. Or if the designers of screens are individual entrepreneurs, they have to be willing, in effect, to pay their own salaries in terms of anticipated entrepreneurial profits. Comparative data on the costs of coding and designing screens is not available, but it is clear that the cost of designing screens and other aspects of the user interface for a computer program is substantial relative to the total cost of development.

In the next issue I will continue with a discussion of the possible difficulties and problems that protection of user interfaces and screen displays might cause. As mentioned earlier, subsequent issues will present conclusions based on these discussions.

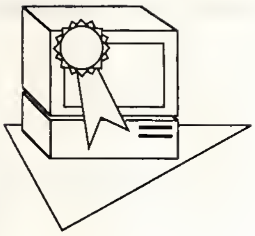
References

1. *Computer*, Vol. 21, No. 9, Sept. 1988, p. 74.
2. *Computer & Software News*, Oct. 12, 1987, p. 1.
3. *Wall Street J.*, Aug. 30, 1988, p. 11, col. 2.
4. Lotus Development Corp. *v.* Paperback Software, Inc., D. Mass., Civ. No. 87-0076K (suit over screen displays and user interface of 1-2-3 spreadsheet computer program).
5. Lotus Development Corp. *v.* Mosaic Software, Inc., D. Mass., Civ. No. 87-0074K (same).
6. M. Dailey, "The 'Look and Feel' of Copyrightable Expression;" *9 Eur. Intel. Prop. Rev. 234*, 235 (1987) (counsel for leading Crosstalk XVI communications computer program attributes commercial success of program "largely" to its screen displays).
7. Whelan Associates, Inc. *v.* Jaslow Dental Laboratory, Inc., 797 F.2d 1222 (3d Cir. 1986), *cert. denied*, 107 S. Ct. 877 (1987).
8. Digital Communications Assoc., Inc. *v.* Softklone Distrib. Corp., 2 U.S.P.Q.2d 1385 (N.D. Ga. 1987).
9. Broderbund Software, Inc. *v.* Unison World, Inc., 648 F. Supp. 1127 (N.D. Cal. 1986).
10. R.H. Stern, "Micro Law: The look, feel, taste, and smell of software," *IEEE Micro*, Apr. 1986, pp. 64-65.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 171 **Medium** 172 **High** 173



New Products

Marlin H. Mickle
University of Pittsburgh

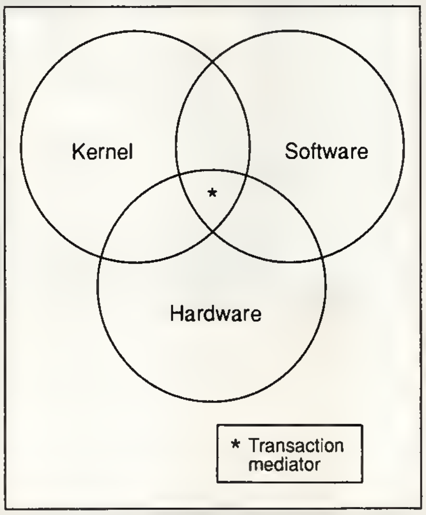
Send announcements of new microcomputer and microprocessor products, and products for review, to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

A holistic response to viruses

According to the company, the Immune System desktop system protects user files from viral attack as well as any other external or internal threats. The system does not allow unauthorized .EXE and .COM files to enter or run on the computer. CRC integrity checks establish the initial conditions of files or programs and run file-integrity procedures.

Hardware and software function together within a complete system to maximize security and prevent problems with coordinating parts of separately secured elements. Communications between hardware, software, and a secured kernel fend off worms, Trojan horses, bombs, or other intrusions by means of a number of layered and interactive proprietary strategies.

Encryption occurs in hardware—rather than software—according to US-government specifications. Hardware includes a 12-MHz 80286 processor with an 80287 socket, 1 Mbyte of DRAM, a 40-Mbyte hard disk, one parallel port, and two RS-232 serial ports. A secured clock prevents the casual user from modifying the system time-date stamp. Because some parts of the security system do not need hard-disk interaction, a section of RAM holds pieces of software in an EPROM-like approach to execute security measures. A set of proprietary chips performs anti-hacker write-protect services.



The Immune System operates in a synergistic fashion that links hardware, software, and a small, highly protected kernel to secure sensitive data.

Programmed security parameters are stored in software along with sensitive data, like audit trails. Utilities execute from within software. Conventional software packages work on the system except for programs that violate security vectors during their regular operations. The company can correct these exceptions.

A secured piece of system RAM interprets all programmed rules and mediates real-time transactions.

User-identification devices include such biometric interfaces as retinal and fingerprint investigations. According to the company, if a data thief succeeds in disturbing hardware security, software recognizes the tampering and shuts down the entire system while securing user data.

Tele-comsec software supports telecommunications security; the system is Hayes-modem compatible. **American Computer Security Industries; $2,995 (Model C2/286-40 with an HGC-compatible monographics card, no monitor); OEM pricing available.**

Reader Service Number 12

Utility merges with editor

The Vedit/SMK package comprises the Seidl Make Utility software-generation system and the Vedit Plus 3.0 multifile programmable text editor. Users automatically generate object files, libraries, and executable files without having to leave the editor. **Compuview Products; $334.**

Reader Service Number 10

Card diagnoses dead PCs

The Power-on Self Test diagnostics card, or Postcard, can perform diagnostics without the support of an operating system. The card lets developers debug systems at various stages of development or troubleshoot in the field. **Award Software.**

Reader Service Number 11

Make your own CASE

A PC-based workbench enables Sylva Foundry MS-DOS users to structure their own CASE tools, methods, techniques, and environments. The workbench helps users embed invisible text and integrate their own trigger programs. **Cadware; from $8,500.**

Reader Service Number 13

More on the business of RISC

One family employs another

The Tek XD88 family includes two graphics workstations, an applications processor, and a file server, all based on the 88000 family. Maximum speeds reach the equivalent of 17 VAX MIPS, 34,000 Dhrystones/s, 16 million single-precision Whetstones, and 12 Mflops. The 88100 processor provides on-chip integer and floating-point multiplication. Four 88200 cache-memory management units contain 64-Kbyte storage. Users can add four more units.

The 3D XD88/30 workstation provides wireframe, shaded solid, and true-color bit-plane configurations with up to 1,310,720 colors. The 2D XD88/20 displays 256 colors. The XD88/01 applications processor can host Tektronix terminals or network stations. The XD88/05 file server includes the applications processor, 1.8 Gbytes of disk storage, and 2 Gbytes of streamer tape. **Tektronix; from $34,950 (XD88/30); from $29,950 (XD88/20); $24,950 (XD88/01); $75,000 (XD88/05).**

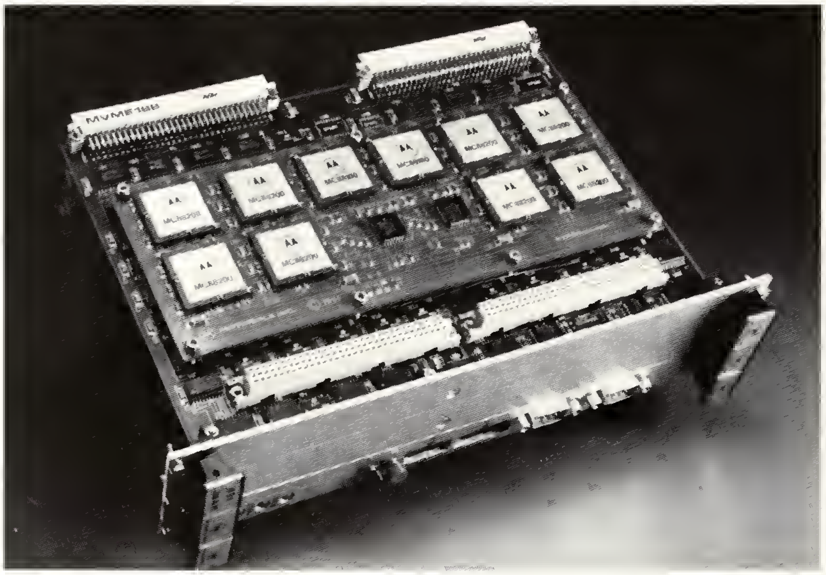
XD88/30 **Reader Service Number 14**
XD88/20 **Reader Service Number 15**
XD88/01 **Reader Service Number 16**
XD88/05 **Reader Service Number 17**

Tools develop 88000 software

Oasys 88K Tools brings RISC applications development to the DEC VAX, the Sun-3 series, the Apple Macintosh II, and the Motorola VME Delta platform. The cross-compiler, assembler/linker, debugger, and simulator also run on 88000-based systems and comply with the Binary Compatibility Standard. **Oasys, Inc.; $15,500 (VAX); $9,400 (Sun, Delta); $4,000 (Mac II).**

VAX **Reader Service Number 18**
Sun **Reader Service Number 19**
Delta **Reader Service Number 20**
Mac II **Reader Service Number 21**

Hypermodule supports VMEbus



The MVME188 RISC multiprocessor quadruple subsystem contains eight MC88200 cache-management units and four MC88100 microprocessors.

The MVME188 series of multiprocessor RISC boards provide up to 60 MIPS of processing power with a 128-Kbyte cache and up to 64 Mbytes of shared main memory in a VME module. Applications include multiuser computation servers, network/communications controllers, and large file and database servers. A VMEbus master/slave interface and a six-unit form factor provide controller compatibility. Software support includes Unix Version 3 and real-time operating systems, development tools, communications, and applications.

MVME188 20-MHz single processors run at 3.75 Mflops (double-precision Linpack), 35,700 Dhrystones/s, and the equivalent of 15 VAX MIPS. MVME188s are compatible with Binary Compatibility Standard software. The company plans dual- and quad-processor versions for this month. **Motorola; $22,950 (single) (100s); $27,200 (dual) (100s); $33,500 (quad) (100s).**

Single **Reader Service Number 22**
Dual **Reader Service Number 23**
Quad **Reader Service Number 24**

Weitek delivers 3D

The XL-8832 floating-point processor offers 20 Mflops of performance for graphics 3520 and 3540 VAXstations. The RISC processor provides single-precision format support in a chip set that consists of the XL-8136 program-sequencing unit, the XL-8137 integer-processing unit, and the XL-3832 floating-point unit. Each device comes in a 144-pin grid array package. Software development tools include an optimizing C compiler, assembler, linker, and debugger. **Weitek; $750 (XL-8832) (1,000s) (OEM).**

Reader Service Number 25

Systems/servers use two processors

Aviion computer system/servers and workstations comprise a family of RISC-based distributed applications architectures. The system/server supports symmetric multiprocessing, incorporates the VME data path, and uses either one or two processors. The system becomes a 250-user server in networking applications that can connect 88000- and non-88000-based workstations. Dual-processor server/systems have a performance rating of 40 MIPS.

Desktop workstations come with 4 million bytes of main memory that is expandable to 28 million bytes. Three data-storage devices can be attached to the system including a 322-Mbyte mass-storage disk and 150-Mbyte magnetic tape cartridge units. Workstation performance reaches 20 MIPS.

The Aviion series uses the DG/UX 4.1 revision of the company's Unix operating system, which is compatible with Unix Version 3, BSD 4.2, and Posix. **Data General; from $52,000 (system/server); $7,450 (workstation).**

Server **Reader Service Number 26**
Station **Reader Service Number 27**

MIPS displays own computations

The RC2030 Riscomputer provides 12 MIPS of speed and 1.8 double-precision Mflops of computational power and plays multiple roles in distributed computing. The desktop workstation can perform either as a host for local-station work groups and X-display stations or as a networked file server. A 16.67-MHz R2000 CPU, an R2010 FPU, and separate 32-Kbyte instruction and data caches support this performance. For large applications, the Riscomputer desktop workstation can support up to 16 Mbytes of main memory and 344 Mbytes of disk storage.

While the RC2030 computes, the RS1210 X-Display station processes the graphics portion of an application. The 16-inch, 105-dpi monochrome display features an X Windows server, a resolution of 1,024 × 1,024, and a 70-Hz, noninterlaced refresh rate. **MIPS Computer System; from $17,000 (RC2030); from $3,200 (RS1210).**

RC2030 **Reader Service Number 28**
RS1210 **Reader Service Number 29**

RISCs support host-coupled graphics

The configurable Adage 200 Color Graphics Processor houses a 192-register file, 25-MHz Am29000 CPU. Virtual Windows technology accompanies a 12 × 24 look-up table for display of 4,096 colors with a 1,280 × 1,024, 60-Hz resolution. Users can combine the VLSI two-board sct with an Egos graphics operating system. Custom options include a 2,560 × 2,048 frame buffer, an expansion VME chassis, and a 25-MHz Am29027 math coprocessor. **Adage, Inc.**

Reader Service Number 30

Unix coprocessor hits 17 MIPS

The Series 400 Personal Mainframe coprocessor makes use of the Motorola 88000 reduced instruction set architecture. The 32-bit machine for computationally intensive applications provides concurrent use of MS-DOS and Unix System V operating systems at maximum speeds of 17 MIPS. This Unix coprocessor system employs the IBM PC AT/XT or PS/2 Model 30/35 as an I/O processor or subsystem for workstations and multiuser configurations.

Total physical memory reaches 20 Mbytes in a 4-gigabyte virtual addressing space. Features include the X Window System Version 11.3 and binary compatibility with other 88000 Unix systems.

A related product, the Personal Mainframe/88SDS software development system, offers C and Fortran compilers. **Opus Systems; from $5,000 (coprocessor) (OEM); from $11,200 (SDS).**

Coprocessor **Reader Service Number 31**
SDS **Reader Service Number 32**

The ins and outs of data

Pick better performance

According to the company, the Pik-fast digitizer table overlay reduces input time to Cadkey microcomputer-based 3D CAD systems by 60 percent. The overlay compresses the number of screen-menu selections required by a mouse into a general, picking-device operation. This function also drives immediate mode commands, on-line calculation, and display-status controls. Other selections are organized by linear-motion, color-key and icon-representation strategies. A centrally located view visualizer and user-definable selection area complete the package. **Jensen Properties (supports Cadkey 3.12; free upgrade with Cadkey 3.5 release).**

Reader Service Number 33

Transducer boasts big buffers

The R1000 waveform digitizer features four channels that each have an 8-bit, 500-KHz A/D converter. Sample data buffers encompass 32 Kbytes per channel. The turnkey peripheral for PCs comes with digital-scope software and drivers for the C, Turbo Pascal, and Basic programming languages. **Rapid Systems; $1,995.**

Reader Service Number 34

Another mouse alternative

The Trackball Plus cursor pointing device emulates Microsoft and Mouse Systems mice and the Bit Pad One digitizing tablet. The six-button trackball also supports Lotus 1-2-3, Word Perfect, and DOS commands through pop-up menus. An RS-232 serial port connects to CAD/CAM environments. **Fulcrum Computer Products; $99.**

Reader Service Number 35

New Products

Company offers "mouseboard"

Keytrak integrates the two most common manual-input devices for PCs: the keyboard and the mouse.

For users who want to use a mouse without abandoning their keyboard, the Keytrak trackball integrates the two. The package is compatible with both Microsoft and Mouse Systems serial mouse drivers. Users can select either the IBM PC AT or XT through a switch under the keyboard. A Y-shaped cable connects the keyboard and serial ports. Three program-dependent mouse buttons reside above the trackball and a two-button mouse is duplicated on the keyboard for flexibility. **Octave Systems; $189.**

Reader Service Number 37

All-in-one graphics touch terminal

The Touchcom GTS combines touch entry, graphics, data communications, and video technologies into one PCB. The board can be used as a terminal or stand-alone system instead of a keyboard. Features include a 768 × 480 × 8-pixel modular graphics memory and a graphics processor with a high-level command language, two RGB video inputs, and resistive touch-screen inputs.

An on-board, 16-MHz 80C186 microprocessor and a modular program memory that accommodates a mix of PROM and RAM of up to 1 Mbyte can substitute for disk memory and an external microcomputer. Software includes the Vrtx real-time operating system, a graphics operating system, an X.25 data-communications package, and a PC-compatible ROM BIOS. **Digital Techniques; under $2,000 (OEMs).**

Reader Service Number 38

Touch screen supports graphics

The Lucas Duralith touch screen provides 1,024 × 1,024-pixel input from front-control panels to applications ranging from medical instrumentation to automated process-control systems.

The Lucas Duralith touch screen combines with custom graphic presentations for insertion into pre-assembled front panels. Users can either mount the panel to rigid sub-panels or to a PCB. When users press two resistive panels together, voltage drops at the intersection and input changes from analog to digital. Design options include interchangeable legends, tactile response switches, and formed overlays. Contact company for custom pricing. **Lucas Duralith.**

Reader Service Number 40

Translator standardizes PCB layout

The IGES Board Station Translator translates graphics and related data files from the Board Station PCB layout system to the International Graphics Exchange Standard. It also translates from standard IGES 4.0 files to Board Station design files. **Mentor Graphics; $15,000 (site license).**

Reader Service Number 36

Data skis cross-country

The international edition of the Xchange PC software conversion program reads MS-DOS country code settings and allows users to display and access accented characters and international punctuation marks. The enhancement "speaks" and displays the local language in use. **Emulation Technologies; $745 (Int'l edition); $150 (upgrade from Xchange).**

Reader Service Number 39

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 180 **Medium** 181 **High** 182

Advertiser/Product Index

FOR DISPLAY ADVERTISING INFORMATION, CONTACT:

Northern California and Pacific Northwest: Roy McDonald Assoc. Inc., 5915 Hollis St., Emeryville, CA 94608; (415) 653-2122.
Jim Olsen, P.O. Box 696, Hillsboro, OR 97123; (503) 640-2011.

Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Suite 100, Torrance, CA 90505; (213) 373-9604.

Southwest: The House Co., 5252 Westchester, Suite 280, Houston, TX 77005; (713) 668-1007.

East Coast: Atlantic Representative Group, 349 Maple Place, Keyport, NJ 07735; (201) 739-1444.

New England: Arpin Associates, 40 Sterling St., Somerville, MA 02144; (617) 625-1777

Europe: Heinz J. Görgens, Parkstrasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G.); phone: (0 21 53) 8 99 88; telex 841(17)2153310=HJG tlx d.

For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE MICRO, 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; phone (714) 821-8380; fax (714) 821-4010.

continued from p. 8

Micro World

Neural Network Research Activities

Institute/Contact	Research specialty	Institute/Contact	Research specialty
University of Stuttgart H. Haken	Pattern recognition	IBM Rome S. Patarnello	Boolean networks Analysis of learning and generalization
Nuclear Energy Research Institute (KFA) Juelich H. Mueller-Krumbhaar	Fuzzy logic Optimization	**Switzerland**	
GMD, Society for Mathematics and Data Processing St. Augustin H. Muehlenbein	Neural network implementation Genetic algorithms	Swiss Federal Insitute of Technology, Zurich O. Kuebler	Image processing
Israel		Swiss Federal Insitute of Technology Lausanne J.D. Nicoud	Neural networks for robotics VLSI implementation Teaching aids
Hebrew University Jerusalem D.J. Amit	Hopfield networks Pattern association Statistic physical methods	CSEM Neucahtel E. Vittoz	Robot control Topology-conserving maps Signal separation
Weizmann Institute Rehovot E. Domany	Associative memories	University of Zurich R. Pfeifer	Artificial intelligence
Italy		Asea Brown Boveri Research Center Baden J. Bernasconi	Algorithms Process automation
University of Rome G. Parisi	Associative memories Hierarchical networks Learning		

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 174 **Medium** 175 **High** 176

Micro View

continued from p. 96

How many pixels do you provide?
Almost a million. Specifically, 1,120 × 832 × 2 bits deep and 94 dots per inch, giving us four shades of gray—black, dark gray, light gray, and white. The four shades render images better than black and white. We would have liked to offer more shades, but we had to keep the price down.

People seem to like color, too, but you decided against it.
Yes, they do—price again forced our decision. The Next cube presently has three vacant slots, so one of the options is going to be—in about a year—a graphics card that drives a color monitor.

Let's talk about software briefly. Why did you select Unix?
It took a little time to sort out the merits of Unix System V v. Berkeley 4.3. We think the two are slowly coming together, but it became clear that Berkeley 4.3 was the right one to be compatible with. It is more widely used in the university environment. We finally selected Carnegie Mellon University's Mach multitasking operating system, which is compatible with Unix 4.3.

How did you make Unix easy to use?
We developed a user interface and development environment that we call Next Step. For the end user this environment masks the complexities of the operating system behind a window-based, graphical workspace. The user can locate and manage files, display the contents of directories, and launch applications and utilities, all without any knowledge of Unix.

What did you do to achieve more performance?
The first step was to select the fastest commercial processors for the portion of the work they are good at. Our group has had long experience with the Motorola 68000 family. We compared it with the other choices that were available

back in 1986 and found it was still a good choice. We decided to go with Motorola's then top-of-the-line microprocessor and memory-management unit, the 68030, a 25-MHz processor capable of 5 MIPS. To its capabilities, we added the MC68882 floating-point coprocessor for mathematical computations and the Motorola DSP56001 digital signal processor to support computation-intensive processes such as sound synthesis.

What is the purpose of the digital signal processor?

The DSP makes it possible to utilize the sound and visual pathways to the brain so learning and communicating are easier. At 10 MIPS the DSP can manipulate waveforms for sound, music, images, real-time analysis of experimental data, and many other phenomena.

What was your second step toward high performance?

Moving data around ordinarily eats up a lot of processor time. So we separated that function from the CPU and put it in specially designed DMA (direct memory access) and data controllers. Our integrated channel processor, for example, manages the flow of data within the system, particularly between main memory, the CPU, and peripheral devices. The optical storage processor handles data flow for the optical disk. Those were two big efforts. Each is contained in one proprietary VLSI chip.

Your decision about main memory is striking. Why did you go to eight megabytes?

At first we thought that four megabytes would be sufficient for someone to run the operating system, the window system, and a small application. But then we realized that pretty soon a user would want a larger application or a second application and then would need more memory. So we decided eight megabytes was the right amount. In addition, a user can add four or eight more megabytes as options.

Of the eight megabytes how much is left for the user after you load the operating system and the other necessities?

After the first application, say the word processor, I think there are about three megabytes left, enough to get a couple more applications in, such as printing. Incidentally, all of the electronics fits on one card in a one-foot cube, powered by a 200-watt supply, so the machine is small, cool, and quiet.

Why did you select a magneto-optical disk and not a large Winchester?

A couple of reasons: removability and reliability. Users can remove the small, 3.5-inch, 256-megabyte optical disk and walk away with all of their files. They can take the files home or on a trip. With a Winchester magnetic drive, however, to do this users have to spend 20 minutes running their files out on a portable tape. And they have to have the tape drives to do it. Second, the optical disk offers higher reliability than the Winchesters because the head is much farther away from the medium. You don't have the head-crash problem found on the Winchesters.

For these reasons we think the optical disk is going to be the storage technology of the 1990s. Still, we offer 330-Mbyte and 660-Mbyte Winchester options for those who want a lot of hard-disk storage.

How long does it take to bring something up from the optical disk?

To launch a program, like the Webster dictionary or the Shakespeare plays (standard issue with the original optical disk), takes 10 or 15 seconds, depending on the program size. Once a program is loaded, finding a particular detail takes only a second or two.

New application programs today are usually distributed on inexpensive floppy disks. With only the optical drive available, how do you plan to get new programs out to the user?

In the short term we expect that software developers are going to use the optical disk as the distribution medium. People who come from the PC marketplace think its $50 price is very expensive. Still, for programs that arrive on seven or eight floppy disks, $50 is not too costly. People who come from the workstation marketplace—and use cartridge tapes costing $30 apiece—don't seem to be bothered by the $50 price. In the long term we expect to see more of the Next machines on networks with users downloading application programs.

Why did you limit your hard-copy output to your own laser printer?

Once you use a laser printer, you get used to its speed, quality, and relative quietness compared to the dot matrix printer. What people really wanted, we found, is a fairly priced laser printer. So we set out to build one at the lowest possible cost, and, at $2,000, we think we have one. Moreover, at 400 dots per inch, it is higher quality than the usual 300-dpi printer.

How did you get screen and print output to match?

Many systems have one language for showing elements on the screen and a separate language for outputting them on the printer. Then a programmer has to write one piece of code for the screen and another piece for the printer. The two outputs don't always precisely match.

We worked with Adobe Systems to develop Postscript to serve both functions. Our original concern was whether Postscript would be fast enough for the screen, but we got it to the point where performance is quite good. So we wound up with one language, Postscript, for both purposes. It makes application development a lot easier.

Earlier you mentioned some trade-offs made to achieve the goals the users wanted. Did you have to trade off anything else?

The machine is more expensive ($6,500 for the educational marketplace; $10,000 commercially) than we would have liked. Over time we hope to be able to trim both those numbers.

Anything else you'd like to add for our readers?

Beyond the design of the product, the three biggest developments we're pleased about are that more than 50 third parties are writing programs for the machine, IBM has licensed the Next Step environment, and Businessland is distributing the product in a second marketplace.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

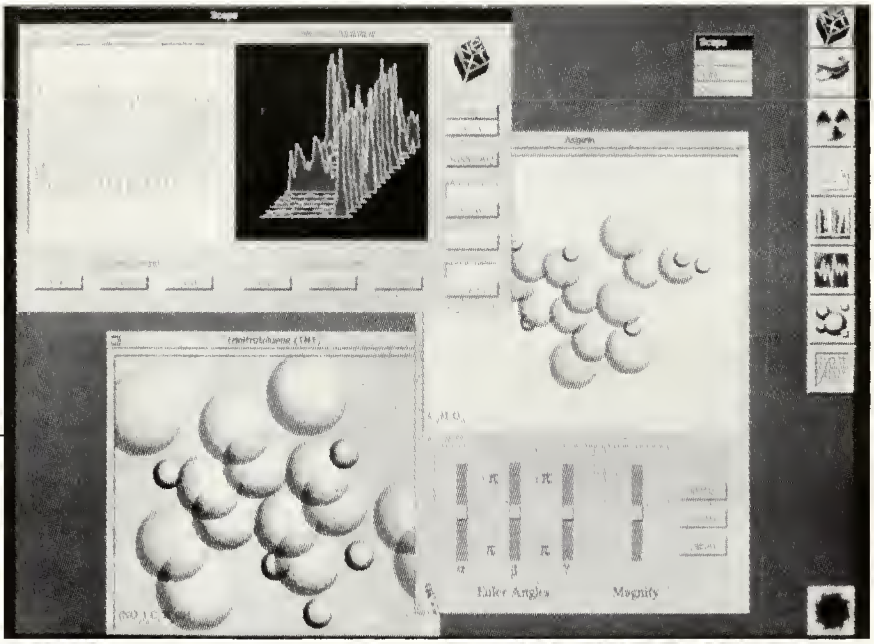
Low 186 **Medium** 187 **High** 188

Department

Micro View

Design choices power the Next wave

Ware Myers
Contributing Editor



The Next workspace reminds us of the Macintosh—yet it's different. Its high resolution (94 dots per inch—the original Macintosh was 70) avoids jaggies as seen in the upper left diagrams and provides texture on the atoms. Its rapid refresh rate (68 Hz) makes rock-solid pictures. The small diagrams at the right margin are icons.

This month Next Inc. begins volume distribution of the Next Computer System from its automated plant in Fremont, California. The Businessland chain of computer stores recently committed to take $100,000,000 of the machines—a cross between a personal computer and a workstation—in the first year. In addition, Steven P. Jobs' latest enterprise will continue to sell directly to the market for which it was originally designed—academe.

As Next views the personal-computer scene, there have been three great waves so far—the Apple, the IBM PC and its clones, and the Macintosh. The fourth wave, the company believes, will be its Next system. How do you go about creating a new wave? Steve Jobs and his long-time associates know something about that. They have done it twice before.

It seems you ask potential users what they would like to have in a system. Then you scour the world of technology for what can be pulled together in time for the new machine. You do all this before anyone else can. You squeeze as many dollars out of the cost as you can. Then make a big splash so the world of users knows what you have. Sound easy? Sure, but it is extremely difficult to make all the pieces fit together.

IEEE Micro asked Next's Richard A. Page how the marketplace requirements fed back into the choices the design team made. Page is vice president of digital hardware engineering and one of six Next founders. Earlier he was instrumental in the initial design of Lisa at Apple and was responsible for the decision to use the Motorola M68000 family in the Lisa and Macintosh computers. We caught him in a brief interlude between trips to Europe and Japan.

What did the marketplace want?

We spoke with potential users at more than a dozen universities—our original target market—in the fall of 1985 and early 1986. They said they needed something more than a personal computer. They liked the power of the workstation, but they characterized it as big, hot, and noisy. They enjoyed the attributes of the personal computer—small, cool, quiet, and reliable.

What else did the users ask for?

They wanted a big enough screen to display a full page of text. They were interested in a Unix-based system, but they wanted it to be easy to use. Of course, they expected a well-designed and consistent workspace. Also, they wanted more performance and more memory—naturally. Learning, or communications in general, involves more than just words. Pictures and sound also help get the meaning across. So we now had more performance requirements.

In addition, they had quantities of information they wanted to have readily available. That meant a lot of mass storage. Some of them were getting accustomed to laser printing, so they wanted that capability, too. They wanted it to be affordable, and they wanted the output to look the same as what they saw on the screen.

That is a big order for a personal computer, or even a workstation. Where did you start?

One of the first decisions we had to make concerned the size of the display. A 15-inch size is too small to show a full page of text. Yet, a 19-inch size is large for a desktop system—users wanted to get away from it. We settled on 17 inches. The determining factor was to have enough pixels so that the screen appears to show a full page, since people relate best to what they see on paper. That is why we are running black letters on a white (or gray) background, too.

continued on p. 94

0272-1732/89/0600-0096$01.00

IEEE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Use the Reader Service Card to obtain information on:

- Membership application—student #203, others #202
- Perodicals subscription form for individuals #200
- Periodicals subscription form for organizations #199
- Publications catalog #201
- Standards working groups list #195
- Compmail+ international electronic mail/database brochure #194
- Technical committee list/application #197
- Chapters lists, start-up procedures—student/regular #193
- Student scholarship information #192
- Awards description/nomination forms #198
- Volunteer leaders/staff directory #196
- IEEE senior member application #204

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

Members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field, plus news, conferences, calendar, interviews, and new products.

Periodicals. The society publishes six magazines and four research transactions. Refer to membership application or request information as noted above.

Conference Proceedings, Tutorial Texts, Standard Documents. The Computer Society Press publishes more than 100 titles every year.

Standards Working Groups. Over 100 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. More than 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.

Chapters. Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

European Office

Payments for Computer Society membership and publication orders are accepted by checks in Belgian, British, German, Swiss, or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by American Express, Diners Club, Eurocard, Master Card, or Visa credit cards.

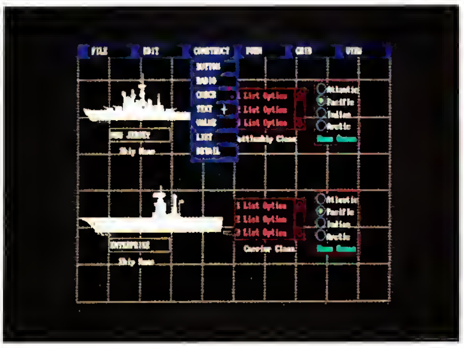
Asian Office

Payments for Computer Society membership and publication orders are accepted by checks in Japanese or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by electronic fund transfer to the Bank of Tokyo, Akasaka Branch, Toza acct. 0767956; the credit receiver is the IEEE Computer Society Headquarters Office. Payment may also be made by American Express, Diners Club, Eurocard, Master Card, or Visa credit cards.

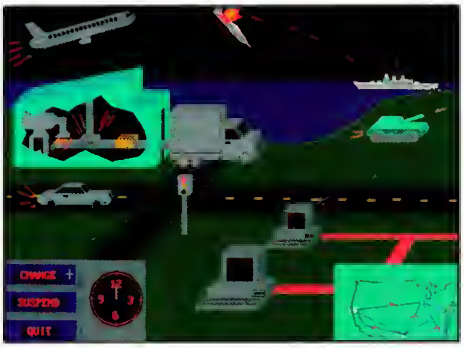
Ombudsman

Members experiencing problems — magazine delivery, membership status, or unresolved complaints — may write to the ombudsman at the Publications Office.

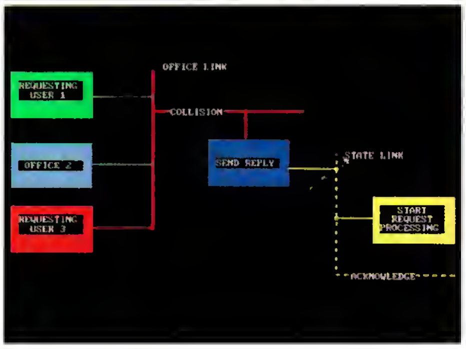
Breakthrough in presentation of simulation results
SIMSCRIPT II.5 with SIMGRAPHICS
Now you see an animated picture of the system



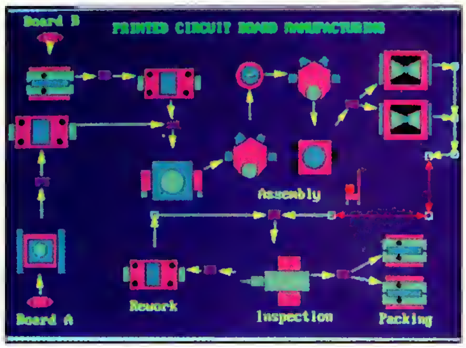
SIMGRAPHICS™ menu builder



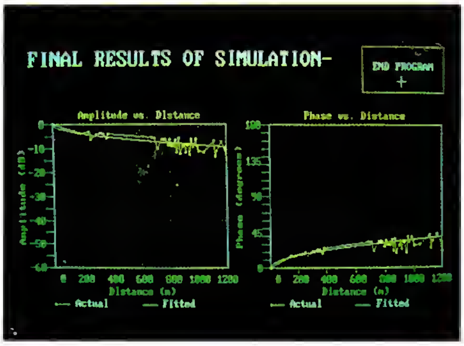
Draw your own icons--or use ours



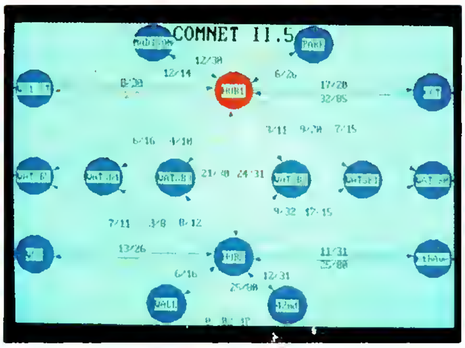
Local Area Network--NETWORK II.5®



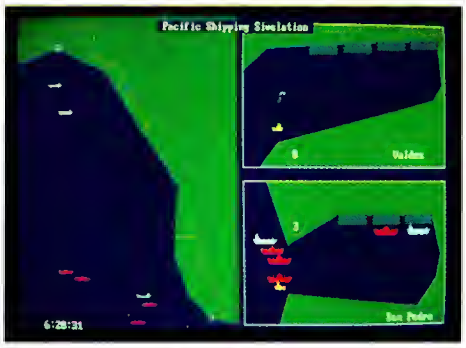
Factory in operation--SIMFACTORY II.5®



Presentation graphics



Communications--COMNET II.5®



Transportation system

SIMGRAPHICS advanced user interface

Free trial and training

See for yourself how simulation results are now easier to understand.

The free trial contains everything you need to try SIMGRAPHICS™ on your computer.

We send you SIMSCRIPT II.5, animated models, and complete documentation. You can build your own model or modify one of ours.

Try the SIMSCRIPT II.5® language, the timeliness of our support, the accuracy of our documentation, and the facilities for error checking-- **everything you need for a successful project.**

For 26 years CACI has provided trial use of its simulation software-- **no cost, no obligation.**

Act now for free training

For a limited time we will also include free training.

For immediate information

Call Hal Duncan at (619) 457-9681, FAX (619) 457-1184. In Europe, call Richard Eve on (01) 528-7980, FAX (01) 528-7988.

Rush information on SIMSCRIPT II.5 with SIMGRAPHICS

Limited offer-return the coupon today and we will also include one free course enrollment worth $950.

☐ Send information on your Special University Offer.

Name ______________________

Organization ______________________

Address ______________________

City __________ State __________ Zip __________

Telephone ______________________

Computer __________ Operating System __________

IEEE MICRO

Return to:
CACI Products Company
3344 North Torrey Pines Court
La Jolla, California 92037
Call Hal Duncan at (619) 457-9681.
FAX (619) 457-1184.

In Europe:
CACI Products Division
Regent House, 89 Kingsway
London, WC2B 6RH, United Kingdom
Call Richard Eve on (01) 528-7980.
FAX (01) 528-7988.